TECNICHE, TRUCCHI, CONSIGLI PER UTILIZZARE SUBITO E AL MEGLIO IL LINGUAGGIO DELLE AZIENDE

# LAVORARE CON JSP

Ivan Venuti

EDIZIONI MASTER

i libri di

# LAVORARE CON JSP

Ivan Venuti

# INDICE

# PREFAZIONE

Questo libro è il seguito di “Imparare JSP” uscito con la rivista ioProgrammo ad Aprile del 2006. Il lettore che non ha letto il libro precedente può tranquillamente far uso di questa guida, purché abbia presenti gli aspetti di base della scrittura di una JSP; in particolare dovrebbe conoscere il linguaggio Java, come installare e configurare Tomcat, sapere cos'è una Web Application, come crearla, configurarla ed installarla, avere una conoscenza, anche minima, delle JSTL e del linguaggio EL. Il presente libro si concentra invece su aspetti considerati “avanzati”, quali un corretto design applicativo, come misurare la performance e come migliorarla, strumenti a supporto della manutenzione del codice e le dovute cautele per le problematiche legate alla sicurezza delle applicazioni. La programmazione di progetti complessi fa spostare l'attenzione dalle funzionalità del codice alla robustezza e chiarezza dell'architettura. In quest'ottica sono presentati dapprima i design patterns ed esempi di un loro uso. Non poteva mancare una panoramica su alcuni dei molti framework che aiutano l'utente nel realizzare applicazioni secondo i pattern più diffusi (oramai il pattern MVC è sinonimo di applicazioni Web!). L'esposizione di tali framework non può essere esaustiva né particolarmente dettagliata (ciascuno di essi avrebbe bisogno di un libro a sé stante!) ma vuole presentare consigli d'uso e modalità di impiego utili ad approfondire uno (o più) dei diversi framework in vista della realizzazione di nuovi progetti. Un intero capitolo è dedicato all'architettura JMX: sono certo che il suo utilizzo renda più semplice la configurazione di una qualsiasi applicazione, a maggior ragione un'applicazione Web che, di solito, è piuttosto complessa e necessita di numerosi parametri e variabili per poter essere resa generale e installabile su un qualsiasi sistema ospite. Inoltre tale architettura per-

mette un monitoring delle risorse usate e delle prestazioni: tutte caratteristiche di fondamentale importanza per siti Web in cui il numero di utenti è potenzialmente molto elevato. Non poteva mancare un capitolo dedicato ai tool che permettono di accedere ai database in maniera semplificata rispetto a JDBC. In ogni capitolo ci sono riferimenti per approfondire e per indagare oltre le nozioni apprese: la curiosità è una caratteristica  essenziale di qualsiasi progettista software che voglia seguire le tendenze in un mondo che evolve in fretta! Ringrazio per i suoi consigli e il suo suppoto mia moglie, Marinella Lizza, prima lettrice di questo mio scritto. Ringrazio anche quanti mi hanno inviato il loro feedback per i libri già pubblicati; spero di averne fatto tesoro per questo volume e per altre pubblicazioni. L'invito è, per tutti voi lettori, di farmi avere le vostre impressioni e i vostri suggerimenti che, sicuramente, terrò in considerazione per le prossibile pubblicazioni. Buona lettura e, soprattutto, auguri di un proficuo riuso delle nozioni qui presentate!

Ivan Venuti
ivanvenuti@yahoo.it

L'autore: Ivan Venuti è laureato in Informatica e lavora come analista programmatore in un'azienda che sviluppa prodotti con tecnologia J2EE. La passione per la programmazione lo ha sempre portato ad approfondire nuove tecnologie, tecniche di sviluppo e framework emergenti. Scrive abitualmente articoli per alcune tra le maggiori riviste di informatica italiane e ha al suo attivo numerosi libri sulla programmazione. Sul suo sito personale, raggiungibile all'indirizzo http://ivenuti.altervista.org, sono messe a disposizione notizie e informazioni aggiornate sulla sua attività professionale.

# DESIGN PATTERN E APPLICAZIONI WEB

Tra le molte tecniche per imparare c'è anche quella che si basa sull'osservare esempi e provare ad applicarli (o ad estenderli) per i propri usi (in inglese si dice "learn by example"). I design pattern partono proprio da questa idea e tentano di formalizzare quali siano "buoni esempi" e quando (e sotto quali condizioni) è possibile applicarli. Questo capitolo offre una loro panoramica e analizza alcuni design pattern particolarmente importanti per la scrittura di applicazioni Web con Java.

## 1.1 UNA NOTA SUGLI ESEMPI

Questa è una nota sugli esempi valida non solo per questo capitolo ma per l'intero libro; quando possibile accanto ad una trattazione teorica sono mostrati uno o più esempi di codice Java. Per comodità tali esempi sono allegati nel CD-Rom della rivista e a cui potete far riferimento senza ridigitare il codice presentato (oppure si può far riferimento al sito *http://cdrom.ioprogrammo.it*).
Gli esempi proposti sono stati realizzati utilizzando Eclipse, un IDE gratuito (e open source) disponibile per il download alla pagina *http://www.eclipse.org/*. Sul CD-Rom sono presenti sia la webapp pronta per essere installata (JspAspettiAvanzati.war) sia i sorgenti all'interno di un progetto di Eclipse (JspAspettiAvanzati_src.zip), facilmente esportabile anche per gli altri IDE. Per far funzionare gli esempi, dopo aver installato Tomcat (versione 5.5) o un altro servlet container, basta copiare all'interno della cartella webapps/ il file JspAspettiAvanzati.war. Gli esempi hanno una pagina iniziale (index.htm) da cui si può accedere in maniera semplice e immediata a tutti gli esempi proposti, sia per eseguirli che per visualizzarne i sorgenti (essi sono suddivisi per capitoli e presentati nell'ordine con cui compaiono nel testo). In **Figura 1.1** la pagina indice, accessibile all'url dove risponde il Tomcat (completa di porta, che di default è la 8080) se

**Figura 1.1**: La pagina indice con tutti gli esempi del libro.

guita dal nome dell'applicazione: /JspAspettiAvanzati/.
Si può notare che in **Figura 1.1**, accanto agli esempi, compaiono due icone. La prima (screenshot dell'esempio) permette di accedere ad un'immagine (statica) che presenta un esempio di esecuzione del programma. È utile in tutti quei casi per cui l'esecuzione del programma non avviene da pagina Web ma da console. Le altre due icone, importate dagli esempi allegati nella distribuzione di Tomcat, permettono, rispettivamente, di mandare in esecuzione l'esempio e di guardarne i sorgenti.

**Attenzione**
I sorgenti sono stati compilati con un JDK 1.5.06 (*http://java.sun.com*) e provati su un Tomcat 5.5.16 (*http://tomcat.apache.org/*). Non ci dovrebbero essere problemi ad usare strumenti più recenti; volendo invece utilizzare una versione di Tomcat precedente (per esempio la 4.1) è necessario intervenire sul file WEB-INF/web.xml (infatti esso si basa sulla specifica delle Servlet 2.4; specifica implementata a partire da Tomcat 5.5). Attenzione al fatto che alcuni esempi (come quelli che fanno uso di JMX) non funzioneranno con JDK precedenti alla versione 1.5.

# 1.2 INTRODUZIONE AI DESIGN PATTERN

I design pattern hanno una storia davvero particolare: un architetto, Christopher Alexander, si pose il problema se fosse possibile, nella costruzione di edifici, identificare delle soluzioni ricorrenti (pattern) e formalizzarle in maniera da usare queste soluzioni come base per la costruzione di un qualsiasi edificio (in questo senso a partire da un ipotetico catalogo di pattern dovrebbe essere possibile costruire un qualsiasi edificio); in questo senso si definisce "pattern" una soluzione ad un problema ricorrente in un determinato contesto. In pratica è la formalizzazione del detto "inutile reinventare l'acqua calda". In realtà l'applicazione in architettura dei pattern non ha mai riscontrato un grosso successo (se parlate di pattern ad un architetto dubito che capisca cosa si intende!); esso ha trovato, viceversa, una enorme applicazione in ambito di progettazione del software a partire dagli inizi degli anni '90, dove vengono usati nel processo di progettazione del software (ecco perché il nome "design patterns"; è molto probabile che un qualsiasi progettista software conosca i design pattern e, per di più, li applichi normalmente nel proprio lavoro!) . Il successo in campo informatico dei design pattern si deve essenzialmente a Gamma, Helm, Johnson e Vlissides (noti come "Gang of Four" ovvero "la banda dei quattro" o, più semplicemente GoF) che nel 1990 hanno catalogato e formalizzato alcuni design pattern applicabili per progetti software.

Da allora è stato formalizzato un numero crescente di design pattern: è possibile, per un progettista software, analizzare il problema e decidere quali tra i numerosi design pattern identificati (e formalizzati) possa portare ad una soluzione ottimale per il problema specifico che si trova ad analizzare.

Esistono interi ambiti, come le applicazioni Web, in cui l'architettura complessiva dell'applicazione può seguire alcuni tra i pattern identificati. Si tenga presente che all'interno di un'applicazione diversi pattern possono essere adottati a diversi livelli di astrazione. Di segui-

to alcuni tra i pattern più utilizzati ed esempi di loro applicazione in ambito J2EE.

### 1.2.1 Vantaggi dei Design Patterns

Il primo, ovvio vantaggio di un design pattern è quello di essere una soluzione particolarmente efficace per risolvere un determinato problema (nessuno si sognerebbe di formalizzare una soluzione inefficace e poco elegante e nessuno si sognerebbe di riusarla!). Il più delle volte, chi si avvicina per la prima volta ai design pattern, rimane piacevolmente sorpreso dal tipo di soluzione e dalla sua eleganza, potenza e versatilità. Però l'applicazione di un design pattern porta, il più delle volte, ad un codice che, una volta esaminato, può risultare complesso e di difficile "decifrazione". Questa presunta complessità viene meno non appena si conoscono i pattern più diffusi; in questo caso anziché "guardare" al codice è più immediato, per i progettisti, riferirsi al pattern. Ecco che diviene un modo di "comunicare concetti" che è più astratto rispetto al codice stesso. Vediamo un esempio di questa presunta complessità. Si guardi la seguente classe:

```
public class EsempioSingleton {
  private static EsempioSingleton istanza
    = new EsempioSingleton();
  private int valore = 0;

  public static EsempioSingleton getIstanza() {
   return istanza;
  }

  private EsempioSingleton() {
  }

```

```
 public int getValore() {
  return valore;
 }

 public void setValore(int nuovo) {
  valore = nuovo;
 }

}
```

Essa può sembrare “strana”; nessun costruttore pubblico è fornito per la classe! Se non si conosce il pattern di riferimento (è il “singleton”) ben difficilmente se ne comprendono il motivo e l’ambito di applicazione...
Proviamo a spiegarlo e tutto sarà molto più chiaro. Per farlo però si deve partire e considerare il problema, non la soluzione!

## 1.2.2 Motivazione ad usare il pattern “Singleton”

Il problema che si vuole risolvere è quello di far sì che una classe abbia una ed una sola istanza durante l’esecuzione di un’applicazione. Questo può essere dovuto a numerosi motivi; vediamone alcuni. Innanzi tutto si può voler “centralizzare” il valore di alcuni contatori o di parametri di configurazione; un modo per farlo è assicurarsi che, in un ambiente dove possono convivere molti thread, ognuno acceda all’unica istanza di una classe (un esempio è una classe che contiene tutti i parametri di configurazione di una Web Application; tipicamente c’è una servlet che viene avviata allo start-up dell’applicazione e che si preoccupa di inizializzare la classe con gli opportuni valori; poi tutte le pagine JSP accedono ai valori della classe).
Un’altra motivazione può essere ricercata nel modo di gestire la me-

moria. Java non possiede un modo per comunicare che un oggetto creato può essere distrutto. Infatti è compito del garbage colletor quello di analizzare periodicamente la memoria e di distruggere oggetti non più referenziati. Questo ha un grosso impatto quando si utilizzano spesso alcune classi: si ha un consumo eccessivo di risorse nella creazione/distruzione di nuovi oggetti. Il singleton permette la creazione di un solo oggetto e offre un modo semplice per tutte le classi di riferirsi alla sua istanza.

## 1.3 CLASSIFICAZIONE DEI DESIGN PATTERN

Si è visto come il pattern Singleton abbia lo scopo dichiarato di far sì che esista al più un'istanza di una classe durante il ciclo di vita di una applicazione. Esso, intervenendo nell'ambito di creazione di oggetti, viene anche classificato come "Creational design pattern". Esistono molti modi di classificare i pattern (in base a cosa fanno oppure in base a chi si applicano) Considerando le categorie di pattern che indicano in quali ambiti il pattern viene utilizzato, si possono sintetizzare questi ambiti (i primi tre si devono alla "Gang of Four", **Tabella 1.1**, i rimanenti tre si devono essenzialmente a M. Grand, **Tabella 1.2**): pattern creazionali (si occupano del modo di creare gli oggetti), pattern strutturali (come si possono comporre classi e oggetti), pattern comportamentali (modo di interagire tra classi e oggetti), patterns fondamentali (chiamati così perché sono la base per la costruzione di altri pattern di livello di astrazione superiore), di partizionamento (partizionano concetti complessi in classi più semplici per migliorare la comprensione del problema) e, infine, sulla concorrenza (relativi a problematiche di gestione della concorrenza). A dire il vero Grand posiziona alcuni pattern in altre categorie rispetto a GoF; altri autori propongono altri tipi di classificazione. In ogni modo l'analisi di tutti questi pattern è rimandata a risorse reperibili sul Web o alla lista di libri presenti in bibliografia.

| Creazionali | Strutturali | Comportamentali |
|---|---|---|
| Singleton | Adapter | Observer |
| Factory | Composite | Chain of Responsability |
| Abstract Factory | Bridge | Strategy |
| Builder | Facade | Visitor |
| Prototype | Proxy | Iterator |
| Object Pool | Virtual Proxy | Mediator |
| | Decorator | Interpreter |
| | Cache Management | Command |
| | Dynamic Linkage | Template Method |
| | | Memento |
| | | State |

**Tabella 1.1** Design patterns identificati dalla Gang of Four.

Iniziamo ora a guardare il pattern MVC: esso può essere considerato un pattern di alto livello di astrazione, che si compone di pattern più semplici (per questo motivo non compare, come altri pattern, nelle classificazioni precedenti). Eppure è il pattern più conosciuto a livello di applicazioni Web e, come tale, merita di essere analizzato per primo.

| Fondamentali | Di partizionamento | Sulla concorrenza |
|---|---|---|
| Delegation | Filter | Single Threaded Execution |
| Interface | Layered Inizialization | Producer Consumer |
| Immutable | | Two Fase Termination |
| Marker Interface | | Read/Write Lock |

**Tabella 1.2** Design patterns identificati da M. Grand.

## 1.4 APPLICAZIONI WEB E IL PATTERN MVC

Il primo aspetto da considerare quando si crea un'applicazione complessa è come essa viene strutturata. Questo porta alla decisione fondamentale di come deve essere il design dell'applicazione stes-

sa (per design si intende la modellazione architetturale). Tra i modelli quello che gode di maggior successo nelle applicazioni Web è l'MVC: Model-View-Controller. Si proverà a scrivere una Servlet che accetta un parametro, dal nome "pagina", che indica il nome della pagina JSP a cui demandare la visualizzazione dei risultati. Se tale pagina esiste, viene mostrato il risultato della sua esecuzione, altrimenti viene intercettata l'eccezione e presentata una form HTML (che, per comodità, apparirà sempre).
In questo modo si è creata una servlet che decide quale JSP deve mostrare il risultato. Questa situazione, generalizzata, è talmente comune che esiste un pattern che la definisce: è il pattern Model View Controller (o semplicemente MVC).
In pratica si cerca di scrivere l'applicazione in maniera da tener distinti e facilmente identificabili i componenti che eseguono i calcoli e le operazioni vere e proprie (parte di Model, assente in questo esempio) dai componenti che si occupano della sola visualizzazione dei dati (View) a quelli che decidono quali operazioni eseguire e chi le deve mostrare (Controller).
In Java tale pattern è implementato (solitamente) da degli EJB o JavaBean per la parte di Model, dalle JSP per la parte di View ed una o più Servlet per la parte del controller (in **Figura 1.2** uno schema logico del pattern).

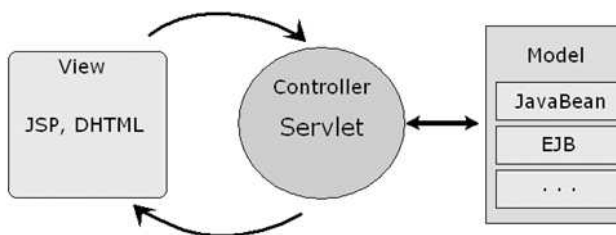



**Figura 1.2:** pattern MVC e ambiente J2EE.

## 1.4.1 Implementare un controller con una servlet

In Java è possibile far rispondere una JSP o una servlet al client e, ad un certo punto, tale pagina/servlet può decidere che una parte della risposta sia mandata da una seconda pagina (o servlet). Questo avviene mediante un meccanismo di inclusione dell'output della seconda pagina. Ecco come potrebbe apparire una servlet che chiede in input il nome di una pagina jsp di cui includere l'output:

```
package it.ioprogrammo.jspAdvanced.pattern;
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class EsempioServletMvcInclude extends HttpServlet{

 public void doGet(HttpServletRequest request,
                                   HttpServletResponse response)
 throws ServletException, IOException{
   String msg = "Specificare un nome di jsp...";
   String nome = request.getParameter("pagina");
   if (nome!=null && !"".equals(nome)){
       RequestDispatcher rd = request.getRequestDispatcher(nome);
       rd.include(request, response);
   }
   PrintWriter pw = response.getWriter();
   scriviHtml(msg, pw);
   pw.close();
 }

 private void scriviHtml(String m, PrintWriter out) throws IOException{
   out.println("<html>");
   out.println("\t<body>");
```

```
    out.println("<br /><hr /><br />\n"+m);
    out.println("\t\t<form>");
    out.println("\t\t\t<input type=\"text\" name=\"pagina\" />");
    out.println("\t\t\t<input type=\"submit\" value=\"Carica\" />");
    out.println("\t\t</form>");
    out.println("\t</body>");
    out.println("</html>");
  }
}
```

Nel file web.xml dell'applicazione si inseriscano i seguenti tag:

```
<servlet>
 <servlet-name>EsempioServletMvcInclude</servlet-name>
 <servlet-class>
  it.ioprogrammo.jspAdvanced.pattern.EsempioServletMvcInclude
 </servlet-class>
</servlet>
 <servlet-mapping>
  <servlet-name>EsempioServletMvcInclude</servlet-name>
  <url-pattern>/EsempioServletMvcInclude</url-pattern>
</servlet-mapping>
```

Accedendo alla url indicata nel file web.xml, la servlet mostra la form in cui chiede il nome di una JSP di cui includere l'output. Ecco una possibile JSP da caricare (includimi.jsp):

```
<html>
 <head>
  <title>JSP minimale</title>
 </head>
 <body>
  Sono una JSP; sono stata invocata in data
```

```
 <%= new java.util.Date() %>
</body>
</html>
```

Se si guarda al codice html generato esso è, ovviamente, non corretto: infatti contiene sia il documento html della jsp sia quello della servlet (Figura 1.3); pertanto per poter usare le include è bene far attenzione che le diverse pagine non inseriscano elementi di apertura/chiusura dei tag html se non dove previsto.
Teoricamente, nel modello MVC, il Controller non dovrebbe mostrare alcun risultato, ma solo coordinare e trasferire opportunamente il controllo. Per esempio si può sostituire l'istruzione precedente rd.include(request, response), con le seguenti:

```
rd.forward(request, response);
 return;
```

**Figura 1.3:** un documento html non corretto (due in uno!).

Queste, infatti, indicano che la risposta alla richiesta viene demandata in toto all'altra pagina (o servlet); è proprio questo il

principio del pattern MVC. Questa seconda versione di servlet è presente, nei sorgenti allegati al libro, in EsempioServletMvcForward.java.

**Attenzione**
È sempre bene specificare una istruzione "return" dopo una forward: non facendolo si ottiene un errore: infatti non sarebbe possibile eseguire ulteriori modifiche all'output una volta trasferito il controllo con un'istruzione forward e questa possibilità è minimizzata imponendo la terminazione del metodo subito dopo la forward. In ogni modo, se si dovessero eseguire modifiche sull'output anche dopo il trasferimento del controllo l'eventuale errore dipende dall'implementazione del Servlet Container (e può variare da versione a versione).

## 1.4.2 Non è sufficiente...

Chiaramente una servlet come quella appena presentata ha ragione di esistere unicamente per dei test. Invece una servlet che realizzi la parte di controller dovrebbe possedere anche altre caratteristiche; per prima cosa dovrebbe poter accedere alla logica di navigazione che è definita esternamente ad essa ma non dentro le pagine stesse di presentazione; per esempio su file XML o su database. Inoltre dovrebbe fornire un meccanismo standard di passaggio dei risultati alle pagine di visualizzazione.
Manca del tutto la parte di gestione del Model; di solito questo avviene applicando un altro pattern (il più diffuso è il pattern "Command"). Come si può intuire realizzare un pattern MVC non è proprio semplicissimo, se si parte da zero. Ma anziché sviluppare una simile servlet ad hoc, insieme a tutto il meccanismo di interazione con la parte model, è consigliabile utilizzare uno dei tanti framework che offrono supporto per realizzare webapp secondo il pattern MVC...

## 1.5 FRAMEWORK E PATTERN

Esistono numerosi framework open source che permettono la realizzazione di applicazioni web Java secondo pattern architetturali che si sono rivelati più adatti a questo tipo di applicazioni. Di seguito l'analisi, ad alto livello, di alcuni di essi.

### 1.5.1 Struts: pattern MVC e molto altro!

Struts è indubbiamente uno dei framework più noti e anche maggiormente utilizzati. La pagina di riferimento del progetto è *http://struts.apache.org/*. Struts nasce come un framework per incoraggiare ed agevolare la scrittura di webapp che facciano uso del pattern MVC (in realtà utilizza un pattern leggermente diverso, chiamato "Model 2"). Attualmente Struts si compone di diversi sotto-progetti, ciascuno dei quali si occupa di una porzione specifica ma che, nell'insieme, compongono il framework ; i sotto-progetti sono: Applications, BSF Scripting, Core, El, Flow, JSF Integration, Plugins, Sandbox, Shale Framework, Taglibs, Tiles.

Ogni sotto-progetto è disponibile per il download sia in versione compilata (binaries; utile per chi vuole usare il framework) che sorgente (utile solo per chi vuole personalizzare ed estendere il framework stesso). Questo permette anche di aggiornare un singolo sotto-progetto alla volta anche se è fortemente raccomandato di eseguire il download dell'ultima release di ciascuno. Ottimi tutorial si trovano alle pagine *http://www.coreservlets.com/Apache-Struts-Tutorial/* ("Demystifying Jakarta Struts"), *http://www.roseindia.net/struts/* ("Struts Tutorials") e *http://www.reumann.net/struts/main.do* ("Struttin' with Struts").

Al momento Struts si divide in due versioni distinte del framework:

- Struts Action Framework
- Struts Shale Framework.

Il primo (Struts Action) è quello che continua a seguire l'idea originaria del framework (idea che si basa sul modello di richiesta/rispo-

sta tipica di una comunicazione secondo il protocollo http). Il secondo, Struts Shal, è basato su componenti ed è pensato per essere integrato con le JavaServer Faces (in pratica esso è l'evoluzione del modello precedente da cui astrae il concetto di chiamata/risposta concentrandosi sugli oggetti che compongono l'applicazione).
È in progetto una release 2 del framework Struts Action; esso si baserà sulle idee di un altro progetto, ora indipendente, chiamato WebWork . I due progetti verranno fusi e diventeranno un unico framework: di Struts rimarrà per lo più la community (molto numerosa), di Web Work le idee fondamentali. Per questo motivo si consiglia fin d'ora, se si vuole imparare un nuovo framework MVC, di iniziare con Web Work.

## 1.5.2 WebWork

WebWork (http://www.opensymphony.com/webwork/) è giunto alla release 2.2.2; il download del file compresso occupa circa 50 Mb e comprende molta documentazione, applicazioni d'esempio e, ovviamente, il framework stesso (anche in formato sorgente).
Una volta scompatto l'archivio compresso, si può aprire un console di comando e posizionarsi nella cartella del progetto e digitare il seguente comando:

```
>java -jar webwork-2.2.2.jar quickstart:showcase
```

In questo modo viene aperto un servlet container con un'applicazione che guida alla conoscenza del framework attraverso degli esempi. Per accedere all'applicazione basta aprire un browser e caricare la pagina *http://localhost:8080/showcase/*. (Figura 1.4).
La cosa particolarmente interessante è che è possibile modificare i sorgenti dell'esempio, che si trovano nella cartella webapps/showcase, e le modifiche vengono rese subito disponibili "a caldo", ovvero senza dover ricompilare o eseguire nuovi deploy! Sul Web è

possibile consultare l'articolo "Hello World the WebWork way" (disponibile alla pagina *http://www.javaworld.com/javaworld/jw-10-2005/jw-1010-webwork.html*) per un esempio di applicazione di WebWork. WebWork permette l'integrazione, al suo interno, di numerosi altri framework; di uno in particolare viene consigliato l'uso: Spring. Esso realizza un diverso tipo di pattern architetturale: l'inversion of control. Di seguito si analizza il significato e il motivo del suo utilizzo.

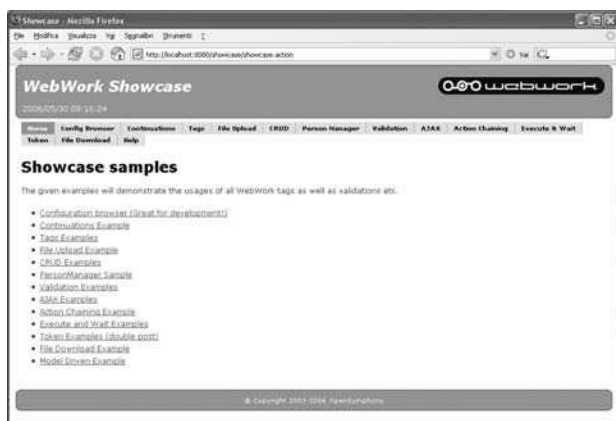



**Figura 1.4:** l'applicazione guida di WebWork.



**Attenzione**
Ovviamente bisogna fare attenzione prima di eseguire l'applicazione guida; se esiste già un server che risponde alla porta 8080 (è il caso di Tomcat!) bisogno prima fermarlo!

## 1.5.3 Spring: Inversion of control

Alcuni framework, ed in particolare quelli che sono anche container per componenti, adottano il pattern "Inversion of control" (IoC). Spring (pagina di riferimento http://www.springframework.org/) è

il framework probabilmente più famoso che adotta (tra le altre cose) l'IoC, ma lo fanno anche HiveMind (*http://jakarta.apache.org/hivemind/*), PicoContainer (*http://picocontainer.org/*) ed Excalibur (*http://excalibur.apache.org/*). Il principio alla base del pattern IoC è quello di disaccoppiare le dipendenze degli oggetti e di chi ne fa uso. Un esempio per chiarire il problema: si supponga di realizzare una classe, Offerta, che possiede un prezzo di vendita (di un ipotetico bene di consumo):

```
public class Offerta{
  private double prezzo;
  public void setPrezzo(double attuale){
    prezzo = attuale;
  }
  public double getPrezzo(){
    return prezzo;
  }
}
```

Supponiamo di realizzare una classe che permette di gestire delle offerte; in particolare che possieda il metodo "prendiMigliore" che non fa altro che prendere un'offerta che ha il miglior prezzo sul mercato. Una possibile implementazione del metodo potrebbe essere la seguente (una iterazione sugli elementi per trovare il prezzo minimo):

```
public class GestioneOfferte{
    OfferteCerca disponibili = new OfferteCerca();
  Offerta prendiMigliore(){
    List tutte = disponibili.listaOfferte();
    Offerta migliore = null;
    for (Iterator i = tutte.iterator(); i.hasNext();) {
            Offerta attuale = (Offerta) i.next();
```

```
            if (migliore==null || attuale.getPrezzo()<migliore.getPrezzo())
                 migliore = attuale;
    }
   return migliore;
  }
}
```

Resta il problema dell'oggetto "disponibili". Si è ipotizzato che esso sia di tipo OfferteCerca; tale classe dovrà possedere un metodo (listaOfferte) che permette di reperire tutte le offerte disponibili. Si può assumere che ricerchi e renda disponibili tutte le offerte di un particolare mercato di riferimento.

Per com'è stato dichiarato il costruttore della classe OfferteCerca, si assume che esso sia una classe concreta. Ma questo crea una dipendenza tra GestioneOfferte e OfferteCerca. In particolare si vorrebbe non dover cambiare OfferteCerca per cercare offerte di un mercato diverso. Quella che si vuole è realizzare un'interfaccia che sia OfferteCerca con il metodo voluto:

```
public interface OfferteCerca{
  List listaOfferte();
}
```

In questo modo si possono realizzare diverse classi, specifiche per i diversi mercati:

```
public class OfferteCercaInItalia{
  List listaOfferte(){
   // realizza
  }
}

public class OfferteCercaInFrancia{
```

```
 List listaOfferte(){
  // realizza
 }
}
```

Resta il problema della classe GestioneOfferte, il cui compito diviene quello di istanziare una particolare classe; ecco come potrebbe essere riscritta:

```
public class GestioneOfferte{
   OfferteCerca disponibili = new OfferteCercaInItalia();
 Offerta prendiMigliore(){
   // come prima
   return migliore;
 }
}
```

Ovviamente è possibile instanziare una qualsiasi classe che implementi l'interfaccia OfferteCerca. Resta da eliminare la dipendenza alla classe concreta specifica (ora è sempre e solo OfferteCercaInItalia). Per farlo ci sono più modi, ciascuno che prende il nome di "dependency injection", e precisamente:

1) Costructor injection: esiste un costruttore che tra i parametri ha anche la classe di cui si vuole eliminare la dipendenza; per l'esempio:

```
public class GestioneOfferte{
   OfferteCerca disponibili;
 public GestioneOfferte(OfferteCerca quale){

   disponibili = quale;
 }
```

```
  // ...
}
```

2) Setter injection: esiste un metodo setter che elimina la dipendenza:

```
public class GestioneOfferte{
  OfferteCerca disponibili;

 public setOfferteCerca(OfferteCerca quale){
  disponibili = quale;
 }
  // ...
}
```

3) Interface Injection: in questo caso si crea una apposita interfaccia per l'iniezione:

```
public interface IniettaOfferteCerca{
 public inietta(OfferteCerca quale);
 }
}
```

ed è la classe GestioneOfferte a implementarla:

```
public class GestioneOfferte implements IniettaOfferteCerca{
  OfferteCerca disponibili;

 public inietta(OfferteCerca quale){
  disponibili = quale;
 }
  // ...
}
```

In tutti i casi si è eliminata la dipendenza tra GestioneOfferte e la classe che realizza OfferteCerca. I framework citati in precedenza offrono diversi meccanismi per realizzare uno o più tipi di dependecy injection; questa soluzione inverte il problema del "controllo su chi realizza la funzionalità" (nell'esempio non è più GestioneOfferte che controlla la classe da usare ma è chi la istanzia a "iniettarne" il contenuto) e per questo viene detto inversion of control.

## 1.6 ALTRI DESIGN PATTERNS NELL'ARCHITETTURA J2EE

All'interno dell'architettura J2EE ci sono molti altri ambiti di applicazione dei design patterns. Ecco, in estrema sintesi, alcuni tra i più significativi per lo sviluppo di applicazioni Web (molti esempi proposti saranno sviluppati nei capitoli seguenti, dove verranno presentati sia il codice che implementa il pattern che l'ambito d'uso più appropriato).

### 1.6.1 Observer

Il pattern Observer definisce una dipendenza uno-a-molti tra oggetti e si applica quando esiste un oggetto dotato di un proprio stato e di un certo numero di altri oggetti interessati a conoscere i cambiamenti del suo stato. In particolare ad essi vengono "notificati" eventuali cambiamenti di stato eliminando la necessità che essi, periodicamente, "interroghino" l'oggetto per sapere se è variato. Spesso si indicano con il termine "eventi" (events) i cambiamenti di stato e con "ascoltatori di eventi" (event listener) gli oggetti interessati a tali cambiamenti. Se si volessero creare delle classi che fanno uso di questo pattern è possibile ricorrere a due classi standard di Java: java.util.Observable e java.util.Observer. La prima è una classe che permette di aggiungere/eliminare listener e metodi per notificare i cambiamenti a tutti gli oggetti che si sono registrati e, volendo, può es-

sere estesa per implementare metodi specifici per notificare i cambiamenti ai listener. Invece java.util.Observer è una interfaccia e deve essere implementata da tutte le classi che vogliono essere listener della classe precedente.
In Java esistono molti tipi di classi Listener predefinite, ciascuna deputata all'ascolto di particolari eventi; al verificarsi di questi eventi, vengono invocati dei metodi dei Listener (i metodi sono definiti da apposite interfacce). Per farlo un Listener deve "registrarsi", ovvero deve notificare ad un gestore di eventi che è interessato a ricevere le notifiche al verificarsi di alcuni eventi. In ambiente grafico è comune lo scambio di notifiche per gli eventi sui componenti grafici (pressione di un pulsante, chiusura di una finestra e così via).
In **Tabella 1.3** una sintesi di alcuni tra i più comuni listener disponibili per le applicazioni web che fanno uso di Servlet (o JSP).

| Listener (nome della classe) | Significato |
|---|---|
| ServletContextListener | Ascolta i cambiamenti sul contesto della servlet (creazione, distruzione) |
| ServletContextAttributesListener | Ascolta i cambiamenti degli attributi (aggiunta, modifica, eliminazione) sul contesto |
| HttpSessionListener | Ascolta i cambiamenti sulle sessioni (creazione, distruzione) |
| HttpSessionAttributesListener | Ascolta i cambiamenti degli attributi (aggiunta, modifica, eliminazione) sulla sessione |

**Tabella 1.3:** Listener relativi alle Servlet (package javax.Servlet).

I listener devono essere dichiarati nel file WEB-INF/web.xml della webapp; cco un esempio di listener:

```
<listener>
 <listener-class>
  nomeDellaClasse
```

```
  </listener-class>
</listener>
```

In seguito verrà fornito un esempio concreto d'uso dei listener (esso verrà utilizzato insieme alla tecnologia JMX per esporre all'esterno il numero di sessioni presenti in una webapp e il numero di errori applicativi avvenuti in essa).

## 1.6.2 Decorator e i filtri http

Un filtro http è un oggetto che può essere associato ad un insieme di url e che possono modificare sia la richiesta HTTP che la risposta. I filtri possono essere messi in cascata e concorrere così alla creazione di una catena di filtri. Ciascun componente della catena può aggiungere informazioni, come specificato nel design pattern Decorator (in pratica ogni nuovo filtro aggiunge, arricchisce le funzionalità dei filtri precedenti). Un esempio di applicazione dei filtri è la realizzazione di meccanismi standard per la gestione delle credenziali di autenticazione. Un esempio di questo tipo sarà sviluppato nel capitolo che tratta la sicurezza nella applicazioni Web (in particolare verrà realizzato un filtro per autenticare gli utenti facendo uso della HTTP basic authentication). Un altro esempio è il seguente: si supponga di realizzare un'applicazione web che fornisce dati cifrati e compressi ad applicazioni esterne che, a loro volta, mandano le loro richieste cifrate e compresse. Un modo per eseguire un design dell'applicazione secondo il pattern decorator è quello mostrato in **Figura 1.5**: la servlet si occupa solo di ricevere richieste e rispondere "in chiaro". Due filtri in entrata si occupano, rispettivamente, di decifrare il messaggio e di scompattarlo. Altri due filtri in uscita effettuano le stesse operazioni in ordine inverso: il primo compatta la risposta, il secondo applica un algoritmo di cifratura.
Qual è il vantaggio di una siffatta applicazione? Che, per esem-

**Figura 1.5:** Esempio di applicazione del pattern decorator.

pio, si può decidere che per un particolare contesto non è più necessario ricorrere alla cifratura dei messaggi. Basterà eliminare i due filtri associati a questa funzionalità. Viceversa nasce l'esigenza di cambiare l'algoritmo di cifratura; ancora basta intervenire solo sui due filtri. Ancora: si vuole introdurre un ulteriore "strato" per eseguire la compressione del testo una volta che esso è già stato compresso e poi cifrato. Basterà aggiungere lo stesso filtro anche agli estremi delle due catene (una di ingresso e una di uscita). Come si vede le possibili estensioni non intaccano tutti gli altri strati, mantenendo concettualmente "pulita" tutta l'applicazione; inoltre, così facendo, la loro manutenzione risulta più semplice.



## 1.7 ANTI-PATTERN: QUELLO DA NON FARE!

Accanto ai design pattern, soluzioni particolarmente eleganti ed efficaci, è stata definita anche una serie di "cose da non fare", cioè di anti-pattern. Può sorprendere il motivo di tale interesse ma, scorrendo alcuni anti-pattern più noti (si veda la pagina *http://www.devx.com/Java/Article/17578 o http://www.devx.com/Java/Article/29162*) ci si può rendere conto del fatto che in molte situazioni, di solito per mancanza di tempo, tali soluzioni inefficaci sono state comunque adottate in qualche progetto passato (e magari ripetute in altri progetti). Pertanto è bene conoscerle non

tanto per "curiosità", ma per scoprire quanti anti-pattern si è soliti usare (credendo, in buona fede, che in fin dei conti siano una soluzione accettabile!) e soprattutto si è soliti ri-usare (errare è umano, ma perseverare è... da informatici pigri!!).

## 1.8 PER APPROFONDIRE

Si è avuto modo di introdurre alcuni pattern. Ma esistono numerosi altri pattern applicabili alle più disparate situazioni. Chiunque faccia della programmazione la propria professione non può non conoscere almeno i principali. Inoltre, iniziando a prendere confidenza con essi, sorge naturale il desiderio di approfondirli, vista la loro potenza e la loro capacità di fornire soluzioni adeguate ai problemi più comuni. In **Tabella 1.4** i principali siti Web per iniziare lo studio. Inoltre non dovrebbe mancare il libro [Gamma et Al., 1994], che diede inizio allo studio dei pattern ed è considerato uno dei libri fondamentali per lo sviluppo di applicazioni professionali.

| Sito Web | Descrizione |
|---|---|
| http://www.patterndepot.com/put/8/JavaPatterns.htm | Download del libro "The Design Patterns Java Companion", interamente dedicato ai pattern architetturali applicati a Java |
| http://www.cim.polito.it/people/guidi/DesignPatternsJava.htm | Un'ottima introduzione, in italiano: "GoF's Design Patterns in Java" |
| http://www.ugolandini.net/PatternsHome.html | Tutorial (in italiano) di livello avanzato<br>Il catalogo dei pattern illustrati nel libro |
| http://java.sun.com/blueprints/corej2eepatterns/ | "Core J2EE Patterns: Best Practices and Design Strategies" |
| http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html | Ottima introduzione all'argomento |

**Tabella 1.4:** Siti Web per approfondire lo studio dei design pattern.

# INTRODUZIONE ALLE JSF

Chi programma pagine dinamiche per il Web sa che una grossa parte del lavoro di programmazione consiste nel gestire opportunamente dati provenienti da form Html e di generare un appropriato risultato a tali invocazioni.
Benché esistano numerosi strumenti per agevolare alcune parti (controllo sui dati utilizzando Javascript lato client, librerie di tag per la visualizzazione uniforme dei campi di inserimento dati e così via), si rischia ogni volta di reinventare l'acqua calda e di cercare soluzioni ad hoc a problemi generali. Inoltre se si programma per client Web "sofisticati" (ovvero i classici browser per PC che hanno plug-in flash, supporto a JavaScript e così via) si utilizzano certi componenti, mentre se lo si fa per smart phone o altri dispositivi mobili se ne usano altri. Per risolvere queste ed altre esigenze è nata una tecnologia ausiliaria all'interno delle JSP, che prende il nome di JavaServer Faces (o, più semplicemente, JSF), il cui scopo è sia quello di generare un'interfaccia grafica (Html, ma non solo!) sia di gestire l'interazione tra client e server per l'immissione, la validazione e la presentazione dei dati. Il tutto viene fatto basandosi su un modello che va aldilà del modello richiesta/risposta imposto dal protocollo http, ma che si concentra sull'uso di componenti software dove i diversi componenti rispondono ad eventi (un modello più simile alla programmazione di interfacce grafiche nei più diffusi linguaggi RAD). Inoltre il ciclo di vita dei componenti è composto da diverse fasi; la "renderizzazione", ovvero il disegno del componente usando costrutti HTML o altro (WML, applet e così via) è solo l'ultima fase. Questo rende possibile, almeno in linea teorica, l'uso degli stessi componenti per dispositivi diversi e facendo in modo che essi vengano resi disponibili con diverse tecnologie (e tutto questo senza cambiare nulla sulle restanti fasi del ciclo di vita, tra cui il design e l'interazione con gli altri componenti!). Di seguito viene analizzato più in dettaglio, con esempi applicativi, l'uso delle JSF.

## 2.1 JAVASERVER FACES

Le JavaServer Faces rappresentano una specifica per applicare componenti per la costruzione sia dell'interfaccia grafica (widgets) sia per gestire, in maniera semplice e semi-automatizzata, il loro stato, astraendosi dal concetto di richieste e risposte HTTP. Questa è un'introduzione minima alla tecnologia: sono fornite le idee di base che ne costituiscono le fondamenta e si offrono link a risorse per approfondire i diversi aspetti. Un libro che copre in dettaglio la tecnologia è, per esempio, "JavaServer Faces in Action" (sono più di 700 pagine! Si veda [Mann, 2005])

| Sito Web | Descrizione |
|---|---|
| http://www.jcp.org/en/jsr/detail?id=127 | Le specifiche della tecnologia |
| http://java.sun.com/j2ee/javaserverfaces/ | Pagina ufficiale della tecnologia (con link a risorse e articoli) |
| http://java.sun.com/j2ee/1.4/docs/tutorial /doc/JSFIntro.html | Introduzione alla tecnologia |
| http://www.horstmann.com/corejsf/ | Pagina del libro "*Core JavaServer Faces*", con capitoli scaricabili liberamente |
| http://www.jsf-faq.com/ | Utile raccolta di FAQ per i problemi più comuni |
| http://courses.coreservlets.com/ Course-Materials/jsf.html | Materiale di studio (in formato PDF) che introduce l'uso della tecnologia |
| http://www.jsftutorials.net/ | Un portale dedicato alle JavaServer Faces |
| http://www.eclipse.org/webtools/jsf/ | Plug-in per Eclipse |
| http://www.ibm.com/developerworks/ library/j-jsf3/ | JSF conversioni e validatori |
| http://developers.sun.com/prodtech/ javatools/jscreator/learning/bookshelf/ | Cinque capitoli tratti dal libro "*Core JavaServer Faces*" di D. Geary e C. Horstmann, edito dalla Sun Microsystems Press |

**Tabella 2.1:** Siti Web per approfondire le JavaServer Faces.

## 2.2 LE IDEE DI BASE

Scrivere un'applicazione per il Web significa realizzare delle pagine in un qualche linguaggio a marcatori (di solito HTML o WML), generate in maniera "dinamica". La navigazione implica l'uso di hyperlink o altri componenti (come bottoni di form); usando dei parametri è possibile personalizzare la visualizzazione di ciascuna pagina.
Benché le Servlet/JSP offrano tutte le caratteristiche per la generazione di queste pagine, è indubbio che offrono ben poco supporto per la creazione di pagine complesse in quanto, spesso, è difficile gestire la parte di visualizzazione (HTML) tenendola separata dalla logica applicativa (scriptlets JSP); questa mancanza porta alla creazione di ambienti di sviluppo integrati (IDE) piuttosto "poveri" e poco sofisticati per la gestione semi-automatica di progetti Web. Le JSF hanno proprio questa funzione: fornire dei componenti standard per la visualizzazione dei dati e permettere l'integrazione con strumenti visuali sia in fase di design grafico che per modellare la navigazione. In realtà le JSF non offrono componenti per la sola visualizzazione, ma rappresentano anche un modo per gestire gli eventi e forniscono dei componenti (lato server) che memorizzano i dati da visualizzare. In pratica si possono distinguere diversi strati "logici":

**1)** componenti per la visualizzazione dei dati ("widget"); si basano su componenti JSP standard, come librerie di tag;
**2)** componenti di memorizzazione dei dati (JavaBean);
**3)** servlet che gestiscono le iterazioni tra i componenti precedenti

A sua volta, la parte di visualizzazione dei dati offre diversi livelli di astrazione. Infatti ci sono dei componenti di interfaccia (User Interface Components) che rappresentano oggetti (anche complessi) per una interfaccia verso l'utente, ma la cui visualizzazione può essere delegata a classi specifiche (chiamate renderers); in questo modo si possono creare kit di renders, ciascuno specializzato per un linguaggio o

per una modalità di visualizzazione (per esempio un kit offre una visualizzazione HTML, un altro WML, un altro ancora SGML; nulla vieta che ci siano kit più complessi, come output per Applet o per altri componenti lato client come Flash o SVG).

## 2.3 UN ESEMPIO MINIMALE

Una pagina JSF minimale è una pagina JSP con la dichiarazione di alcune librerie di tag; ecco un esempio di ossatura di pagina JSF che contiene la libreria core e quella html (la pagina è compresa negli esempi con il nome di primaJsf.jsp):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<html>
 <head>
  <!-- solita intestazione html -->
 </head>
 <body>
  <!-- contenuto da visualizzare -->
 </body>
</html>
</f:view>
```

Si possono osservare, accanto agli usuali tag HTML (compresi commenti che servono unicamente da "marcatori" e che vanno sostituiti con del contenuto reale), l'indicazione di due librerie di tag (la core e l'html) e l'uso di un primo tag, chiamto "view" (della libreria core). Esso racchiude la parte di codice HTML da mostrare al browser del client. L'altra libreria di tag permette di specificare i diversi componenti html, come le form e, al suo interno, i diversi componenti di input.

Affinché la pagina sia riconosciuta come JSF è necessario intervenire sul file WEB-INF/web.xml; infatti è necessario che le pagine JSF vengano gestite da una servlet, più precisamente da javax.faces.webapp.FacesServlet. Per questo sono necessari due interventi: il primo consiste nel dichiarare la servlet:

```
<servlet>
 <servlet-name>Faces Servlet</servlet-name>
 <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>
```

il secondo consiste nell'assegnare alla servlet un insieme di url in cui essa viene invocata per interpretare le pagine JSF. Essenzialmente si potrebbero assegnare pattern in cui la prima parte è un prefisso per tutte le JSF (come /faces/*; è il caso standard):

```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

oppure assegnarle una nuova estensione (per esempio .faces):

```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

Si noti che se la pagina è /pagina.jsp, accedendo alle url /faces/pagina.jsp (nel primo caso) o /pagina.faces (nel secondo) viene automaticamente invocata la servlet che interpreta la pagina come JSF!
Ritorniamo ora al sorgente della pagina JSF e facciamo in modo che

mostri del contenuto; per esempio al posto del commento (contenuto da visualizzare) si possono iniziare ad inserire dei componenti JSF. In JSF alcuni componenti sono dei semplici contenitori, come panelGrid (esso si "trasforma" in una tabella html), altri componenti permettono di generare dell'output; un esempio è il tag outputText (in pratica stampa un testo) e graphicImage (per mostrare un'immagine):

```
<h1>Prima pagina JSF</h1>
<f:view>
 <h:panelGrid columns="3" border="1">
  <h:outputText value="del testo"/>
  <h:graphicImage url="immagini/ivan.jpg"
   alt="una figura"/>
  <h:outputText value="del testo"/>
 </h:panelGrid>
</f:view>
```

Salvata la pagina come "esempioJsf_01.jsp", la si può interrogare con l'url http://localhost:8080/JspAspettiAvanzati/faces/esempioJsf_01.jsp (si noti l'uso della cartella /faces/ nell'url: è quella definita nel file web.xml per le JSF!). Il risultato, a livello di sorgente html, è il seguente (mentre in **Figura 2.1** è mostrato quello che appare nel browser):

```
<h1>Prima pagina JSF</h1>
<table border="1">
<tbody>
<tr>
<td>del testo</td>
<td><img src="immagini/ivan.jpg" alt="una figura" /></td>
<td>del testo</td>
</tr>
</tbody>
</table>
```

**Figura 2.1:** il risultato di /faces/esempioJsf_01.jsp

Chiaramente si possono "comporre" in qualsiasi modo i componenti; ecco la stessa pagina leggermente modificata che "annida" gli stessi componenti visti in precedenza (esempioJsf_02.jsp, il risultato è mostrato in **Figura 2.2**).

```
<h1>Seconda pagina JSF</h1>
<f:view>
 <h:panelGrid columns="3" border="1">
  <h:outputText value="del testo"/>
  <h:panelGrid columns="1" border="1">
   <h:outputText value="del testo"/>
   <h:graphicImage url="immagini/ivan.jpg"
    alt="ancora una figura" />
   <h:outputText value="del testo"/>
  </h:panelGrid>
  <h:outputText value="del testo"/>
 </h:panelGrid>
</f:view>
```

Che accade se si inserisce un attributo qualsiasi su di un tag? Per esempio,si provi a inserire l'attributo "pippo" per graphicImage:

**Figura 2.2:** il risultato di /faces/esempioJsf_02.jsp.

```
<h:graphicImage url="immagini/ivan.jpg"
  alt="ancora una figura" pippo="test" />
```

In teoria sul codice HTML un attributo non riconosciuto verrebbe ignorato. Invece provando a richiedere la pagina JSF modificata si ottiene un errore.

```
org.apache.jasper.JasperException: /esempioJsf_02.jsp(14,4)
 Attribute pippo invalid for tag graphicImage according to TLD
```

In pratica l'attributo non viene riconosciuto e la pagina non viene mostrata. Questo fa capire come le JSF possono validare gli attributi per i diversi tag ed evitare errori di digitazione e/o sintattici (il motivo è quello mostrato nel messaggio: esiste un TLD, ovvero una specifica formale di definizione in linguaggio XML, che permette di validare i tag utilizzati). Esistono molti altri tag; una lista di tutti i tag standard JSF è presente alla pagina http://www.horstmann.com/corejsf/jsf-tags.html. Inoltre possono essere utilizzate librerie aggiuntive di tag che estendono le funzionalità predefinite e nulla vieta di creare librerie di tag personalizzate adatte ai propri scopi.

# 2.4 NAVIGAZIONE E MANAGED BEAN

Negli esempi appena visti mancano due aspetti importanti, sempre presenti nelle applicazioni "reali": il primo riguarda i dati "persistenti". Dove vengono salvati i dati provenienti da form html (e quindi riletti)? In secondo luogo, come viene modellata la parte di navigazione (submit di form, pulsanti di navigazione, hyperlink)? Proviamo a rispondere a queste domande, partendo dall'ultima. Si crei un nuovo file JSF (esempioJsf_01.jsp) contenente la seguente form:

```
<f:view>
<h:form id="nomeForm">
 <h:inputText id="colore" value="" />
 <h:commandButton id="comando" type="submit"
  value="Mostra" action="mostra" />
</h:form>
</f:view>
```

La parte di navigazione vera e propria qui è inserita nel tag commandButton; infatti è lì che si trova l'attributo action="mostra". Esso indica che l'azione, che segue la pressione del bottone di comando, è chiamata "mostra"; pertanto c'è un nome logico e non una vera e propria url. Che accade quando avviene tale azione? Nelle applicazioni JSF tutti dettagli sulla navigazione sono inseriti unicamente in un file di configurazione specifico (dove, in pratica, è inserita la mappa di navigazione completa per l'applicazione; ciascun elemento della mappa specifica i punti di partenza e quelli di arrivo della mappa). Il file di configurazione è WEB-INF/faces-config.xml; al suo interno ciascun "passo" della navigazione è inserito nel tag navigation-rule:

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
```

```
 "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
<navigation-rule>
 <!-- regole di navigazione -->
</navigation-rule>
</faces-config>
```

Nelle regole di navigazione si inseriscono le pagine di partenza e quelle di arrivo; per queste ultime si deve indicare anche l'azione che porta a seguire il "passo" descritto; ecco come modellare il fatto che a partire dalla pagina esempioJsf_03.jsp si arriva a esempioJsf_04.jsp quando l'azione è "mostra":

```
<navigation-rule>
 <description>
  Dalla pagina di input a quella di visualizzazione
 </description>
 <from-view-id>/esempioJsf_03.jsp</from-view-id>
 <navigation-case>
   <from-outcome>mostra</from-outcome>
   <to-view-id>/esempioJsf_04.jsp</to-view-id>
 </navigation-case>
</navigation-rule>
```

Ecco come si potrebbe scrivere la pagina esempioJsf_04.jsp (pulsante di ritorno):

```
 <f:view>
 <h:form id="nomeForm">
  <h:commandButton id="comando" type="submit"
   value="Da capo..." action="indietro" />
 </h:form>
</f:view>
```

ed ecco come modellare l'azione di "indietro" con una nuova regola (distinta dalla precedente):

```
<navigation-rule>
 <description>
  e ritorno
 </description>
 <from-view-id>/esempioJsf_04.jsp</from-view-id>
 <navigation-case>
   <from-outcome>indietro</from-outcome>
   <to-view-id>/esempioJsf_03.jsp</to-view-id>
 </navigation-case>
</navigation-rule>
```

Eppure una navigazione di questo tipo (sempre e solo verso una specifica pagina) può essere una situazione troppo semplificata; si pensi a tutti i casi in cui si possono verificare errori nei dati inseriti e l'azione conseguente è il reload della pagina contenente la form; ecco come modellare una siffatta situazione usando l'azione "errore":

```
<navigation-rule>
 <description>
  Dalla pagina di input a quella di visualizzazione
 </description>

 <from-view-id>/esempioJsf_03.jsp</from-view-id>
 <navigation-case>
   <from-outcome>mostra</from-outcome>
   <to-view-id>/esempioJsf_04.jsp</to-view-id>
 </navigation-case>

 <navigation-case>
   <from-outcome>errore</from-outcome>
```

```
  <to-view-id>/esempioJsf_03.jsp</to-view-id>
 </navigation-case>
</navigation-rule>
```

Come si vede la navigation-rule è la stessa inserita precedentemente, ma con un navigation-case aggiuntivo!
Come decidere la validità dei dati e, soprattutto, quale entità si occupa di memorizzarli tra una richiesta e l'altra? La parte dei dati è, come detto, gestita da dei JavaBean. Essi si richiamano con delle particolari espressioni in EL (Expression Language) all'interno delle pagine JSF; tali espressioni hanno la seguente forma:

```
value="#{gestoreBean.qualeColore}"
```

Questo equivale ad accedere ad una proprietà chiamata "qualeColore" di un JavaBean (riferito per nome: "gestoreBean"); la proprietà del JavaBean può essere sia in lettura che scrittura (anche entrambi); se è in lettura, il JavaBean deve possedere un metodo getQualeColore, se in scrittura, un metodo setQualeColore, se è sia in lettura che scrittura (caso più comune), entrambi i metodi:

```
package it.ioprogrammo.jspAdvanced.jsfbean;
public class gestore {

 private String qualeColore ="EEEEEE";

 public String getQualeColore() {
   return qualeColore;
 }
 public void setQualeColore(String qualeColore) {
   this.qualeColore = qualeColore;
 }
}
```

Questo bean va dichiarato; per farlo si usa sempre il file WEB-INF/faces-config.xml ma, questa volta, si usa la sezione managed-bean (sezione che precede navigation-rule):

```
<managed-bean>
 <description>Un bean di gestione</description>
 <managed-bean-name>gestoreBean</managed-bean-name>
 <managed-bean-class>

  it.ioprogrammo.jspAdvanced.jsfbean.gestore
 </managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Come si può vedere è necessario definire il nome del bean visibile a livello JSF (gestoreBean) insieme alla classe che implementa il bean e, infine, specificando il livello di visibilità del bean stesso (in questo caso la sessione).
Modificando il file esempioJsf_03.jsp si può fare in modo che il valore dal campo di input assuma il valore di qualeColore del bean:

```
<h:inputText id="colore"
 value="#{gestoreBean.qualeColore}"
/>
```

In questo modo, appena si apre la pagina, viene mostrato il valore di default assegnato alla proprietà. Premendo sul pulsante "Mostra" e poi ritornando indietro premendo su "Da capo..." ci si accorge che viene mostrato nel campo anche l'eventuale valore modificato (si ricorda che #{gestoreBean.qualeColore} è una notazione che indica sia la lettura del valore che la successiva scrittura!). Non resta che inserire, nella pagina esempioJsf_04.jsp, una scritta colorata con il colore scelto:

```
<h:outputText
 style="color:##{gestoreBean.qualeColore}"
 value="Colore scelto" />
```

Il risultato, supponendo di lasciare il valore di default "EEEEEE" nella casella di inserimento, mostrerà il seguente codice html:

```
<span style="color:#EEEEEE">Colore scelto</span>
```

Ma che accade se l'utente non digita un codice di colore valido? E soprattutto, come (e quando) si controlla la sua validità?

## 2.5 VALIDARE L'INPUT

Validare l'input di solito comporta due azioni: analizzare se l'input è corretto ed, eventualmente, notificare all'utente eventuali errori (possibilmente con dei messaggi significativi per far comprendere che tipo di errore si è verificato!). Le JSF demandano la validazione dell'input a particolari componenti, chiamati "validatori" (validators). Esistono validatori standard ma anche la possibilità di crearne di personalizzati. Di seguito una panoramica delle diverse possibilità.

### 2.5.1 Errori gestiti con pagine diverse

Già in precedenza si è visto come modellare, attraverso la navigazione, il passaggio da una pagina ad altre due in base a due navigation-case, uno associata a "mostra" l'altro ad "errore". Però non si è visto come far seguire le due strade. Un modo è quello di associare una action specifica al pulsante di navigazione della pagina esempioJsf_03.jsp:

```
<h:commandButton id="comando"
 type="submit" value="Mostra"
 action="#{gestoreBean.operazione}" />
```

In questo caso si può vedere che l'action si riferisce al gestoreBean introdotto in precedenza e ad una nuova proprietà: "operazione". Questa può essere modellata come il seguente metodo nella classe it.ioprogrammo.jspAdvanced.jsfbean.gestore:

```
public String operazione(){
  if (tuttoOk())
     return "mostra";
  else
     return "errore";
}
```

dove tuttoOk può essere un metodo che verifica la correttezza dell'input; quando l'input è corretto, l'operazione restituisce "mostra"; se non è corretto, restituisce la stringa "errore" (si noti che i valori sono proprio quelli modellati nella specifica dei due casi di navigazione). Il metodo tuttoOk, in questo caso, potrebbe semplicemente indicare se qualeColore è un numero esadecimale:

```
private boolean tuttoOk(){
 try {
  Integer.valueOf( qualeColore, 16);
  return true;
 }catch(NumberFormatException e){
   return false;
 }
}
```

L'esempio mostra come cambiare l'azione in base al fatto che l'input sia corretto oppure no. In realtà è stata fatta una forzatura: infatti è bene applicare tale meccanismo non per la validazione dell'input, ma solo per seguire strade diverse in base ai risultati ottenuti eseguendo l'operazione. Si pensi al tentativo di inserimento di un nuovo record sulla base dati: l'input è stato verificato ed è risultato sintatticamen-

te corretto, ma durante la scrittura c'è stato un errore (per esempio il database non è in linea); in questi casi non è stato l'utente a scrivere qualcosa che non doveva, ma è l'operazione che non è andata a buon fine. Per distinguere questo caso "anomalo" da quello normale (di inserimento con successo) si può usare la strategia di cambiare l'azione. Invece, per la validazione sintattica e/o semantica, è bene seguire altre strade. Vediamo quali...

## 2.5.2 Valori obbligatori e messaggi

Innanzitutto la prima verifica è quella che tutti i campi obbligatori debbano essere inseriti. Per farlo c'è l'attributo "required" sui tag di input che, per i valori obbligatori, va impostato a true; nel nostro esempio:

```
<h:inputText id="colore"
 value="#{gestoreBean.qualeColore}"
 required="true" />
```

Se ora si prova a premere il pulsante di comando quando non c'è alcun testo digitato, sembra che non accada nulla in quanto viene ricaricata la stessa pagina. È bene rendere esplicito con un messaggio ciò che ci si aspetta: l'inserimento di un valore nel campo. Per farlo si inserisca il seguente codice al termine della form:

```
<h:message id="error"
 for="colore"
 style="color:#ff5555" />
```

Questo tag indica di scrivere un messaggio quando c'è una situazione di errore sul tag con id uguale a "colore" (si noti l'id e l'attributo for: il primo indica che gestisce l'errore, il secondo specifica per quale componente). Rifacendo clic sul pulsante quando non c'è testo nel-

la casella di input, ora viene mostrato un messaggio che avvisa l'utente che ci si aspetta un valore sul campo (**Figura 2.3**).

## 2.5.3 Validatori standard

Accanto al controllo che venga valorizzato il campo, si possono validare anche i valori inseriti prima. Per esempio ecco come far sì che venga controllata la lunghezza del testo (un colore valido per l'html è composto da stringhe di esattamente 6 caratteri):

**Figura 2.3:** Messaggio di errore quando non si scrive alcun colore.



```
<h:inputText id="colore"

 value="#{gestoreBean.qualeColore}"
 required="true">

   <f:validateLength minimum="6" maximum="6"/>
</h:inputText>
```

In questo caso se la lunghezza inserita è diversa appare un diverso messaggio di errore (**Figura 2.4**).
Esistono anche altri validatori standard (**Tabella 2.2**) ed è possibile definirne di personalizzati.

**Figura 2.4:** Messaggio di errore per valori di lunghezza diversa da 6.

## 2.5.4 Validatori personalizzati

Per l'esempio se si volesse realizzare un ulteriore validatore, che è quello che in principio è stato realizzato nella classe gestore: il metodo tuttoOk(). Ecco come si dovrebbe riscrivere la JSF:

```
<h:inputText id="colore"
 value="#{gestoreBean.qualeColore}"
 required="true"
 validator=#{gestoreBean.tuttoOk}
 >
   <f:validateLength minimum="6" maximum="6"/>
```

| Validatore | Descrizione |
|---|---|
| validateLength | Valida la lunghezza (attributo "minimun" per specificare il numero minimo di caratteri e "maximum" per specificare quello massimo). |
| validateDoubleRange | Valida che ci sia un numero che possa essere convertito in Double. Se sono specificati gli attributi minumun/maximun controlla, rispettivamente, che il valore sia maggiore o uguale a mininum e minore o uguale a maximun |
| validateLongRange | Come il validatore precedente ma con valori di tipo Long |

**Tabella 2.2:** Validatori standard.

```
</h:inputText>
```

Però il metodo che abbiamo già realizzato nel bean non funziona; infatti i validatori devono avere questa signature:

```
public void nomeMetodo(FacesContext facesContext,
  UIComponent component, Object nuovoValore)
 throws ValidatorException
```

In pratica all'interno del metodo eventuali situazioni di errore (ovvero input non valido) vengono segnalate sollevando una eccezione di tipo ValidatorException a cui si passa un oggetto di tipo FacesMessage che contiene la stringa da visualizzare. Ecco, nel caso specifico, come si realizza tale metodo (in Figura 2.5 un caso di errore segnalato da questo validatore):

```
public void tuttoOk(FacesContext facesContext,
   UIComponent component, Object nuovoValore)
   throws ValidatorException{
 if (!tuttoOk())
  throw new ValidatorException(
    new FacesMessage(

     "Codice di colore non valido."+
     " Deve essere un numero espresso in base 16."));
}
```

## 2.5.5 Messaggi JSF e bundle Java (i18n)

Si può notare che tutti i messaggi di errore dell'applicazione (Figura 2.3 e 2.4) sono in inglese. Questo avviene perché le JSF non hanno messaggi di errore in italiano e di default (in mancanza della lingua specifica), essi sono in inglese. Per fortuna è possibile sia modificare i

**Figura 2.5:** Messaggio di errore per valori di lunghezza diversa da 6.

messaggi standard sia impostare messaggi per altre lingue. Per farlo è necessario comprendere come Java tratti l'internazionalizzazione delle applicazioni (il termine inglese "internazionalization" designa questa problematica e, visto che la parola, al contrario di molte altre inglesi, è piuttosto lunga, di solito la si indica con "i18n": dove "i" ed "n" sono, rispettivamente, la prima e l'ultima lettera della parola, 18 è il numero di lettere intermedie") e, in particolare, conoscere il meccanismo dei "resource bundle". Procediamo con ordine.
Java ha sempre posto molta attenzione al problema della creazione di programmi che possano essere facilmente tradotti in lingue diverse da quella originale. Il primo problema da risolvere è quello di creare un meccanismo per cui tutte le risorse che possono essere personalizzate siano separate dal resto dei programmi (in pratica i messaggi, le date, eventuali immagini possono cambiare, la logica dei programmi no). Per questo sono stati introdotti i resource bundle: essi permettono la memorizzazione di informazioni che, fisicamente, sono poste al di fuori dei file .class (o .java). I resource bundle possono essere, a loro volta, realizzati in molti modi diversi (altri file .java da compilare, file testuali, e così via) ma tutti hanno una caratteristica: ogni risorsa è identificata da due parti: una chiave e un valore. La chiave è usata nei file sorgente per referenziare la risorsa, il valore è assegnato, di volta in volta, per ogni lingua. Resta da capire come si identifica la lingua. Questa viene desunta dall'ambiente di esecuzio-

ne (programmi stand-alone analizzano le proprietà del sistema, le applicazioni web analizzano lo header http che indica la lingua principale usata nel browser e così via). I resource bundle usano una convenzione: la prima parte del file è fissa (per esempio "risorse-da-tradurre") poi possono seguire due codici: il primo si riferisce alla lingua ("it" per italiano, "en" per inglese etc.) il secondo alla nazione ("it" per Italia, "us" per Stati Uniti etc). Entrambi i codici sono opzionali, ma se è presente il secondo lo deve essere anche il primo. Ecco, per esempio, un insieme valido di nomi di resource bundle (l'estensione indica che sono file di properties):

```
risorse-da-tradurre.properties
risorse-da-tradurre_it.properties
risorse-da-tradurre_it_it.properties
risorse-da-tradurre_en.properties
risorse-da-tradurre_en_us.properties
risorse-da-tradurre_fr.properties
```

Il principio è quello di utilizzare sempre il resource bundle più specifico che si possa applicare; quello senza codici è quello di default. Per esempio se si collega un utente dall'Italia, esso avrà associato il file risorse-da-tradurre_it_it.properties. Se si collega un utente Svizzero di lingua italiana, avrà associato il file risorse-da-tradurre_it.properties. Se si collega dalla Francia sarà risorse-da-tradurre_fr.properties. Lo stesso file (risorse-da-tradurre_fr.properties) vale anche per chi si collega dalla Svizzera e usa la lingua francese. Se si collega un utente Austriaco si userà il file risorse-da-tradurre.properties etc. etc.
Anche le JSF adottano i resource bundle. Per accorgersene basta aprire il file jsf-impl.jar e fare una verifica spostandosi nella cartella javax/faces; in essa ci sono i seguenti file:

```
Messages.properties
Messages_de.properties
```

```
Messages_es.properties
Messages_fr.properties
```

Il file di default è inglese (basta aprirlo per sincerarsene), gli altri traducono i termini in, rispettivamente, tedesco, spagnolo e francese. Manca l'italiano! Poco male: basta copiare uno di questi, per esempio Messages.properties, e rinominarlo in Messages_it.properties. Editarlo (con qualsiasi editor text plain), salvarlo e posizionarlo in una posizione opportuna. Tale posizione può essere o all'interno del JAR stesso, accanto agli altri (non è proprio una buona idea) o sotto WEB-INF/classes/javax/faces (scelta consigliata e seguita negli esempi del libro). Ora basta eseguire l'applicazione con un browser in cui è settata la lingua italiana per visualizzare i nuovi messaggi inseriti (Figura 2.6).

## 2.5.6 Messaggi applicativi

Con lo stesso principio dei messaggi di default delle JSF è possibile creare una qualsiasi applicazione Java e, ovviamente, inserire proprie pagine JSF "localizzate". Per farlo bisogna indicare nel file WEB-INF/faces-config.xml quali sono le lingue supportate; ecco, per esempio, come indicare che sono supportate le lingue inglese, spagnolo e italiano e che quest'ultima è la lingua di default:

```
<application>
```

**Figura 2.6:** Messaggio di errore (ora in italiano!).

```
<locale-config>
 <default-locale>it</default-locale>

 <supported-locale>en</supported-locale>
 <supported-locale>es</supported-locale>
 <supported-locale>it</supported-locale>
</locale-config>

<message-bundle>
MyJsfMessages
</message-bundle>
</application>
```

Il tag “message-bundle” definisce il nome dei file di bundle. Tali file dovranno avere il nome come quello inserito nel tag, più l’usuale convenzione sulle lingue. In questo esempio ci dovranno essere:

```
MyJsfMessages.properties
MyJsfMessages_en.properties
MyJsfMessages_es.properties
MyJsfMessages_it.properties
```

Ognuno contiene i messaggi per la lingua specificata; essi vanno posti nel classpath dell’applicazione; il posto più indicato è la cartella WEB-INF/classes/, ma nulla vieta di porli in una qualche libreria JAR sotto la cartella WEB-INF/lib/. È preferibile, per non inserire tutti i file nella stessa cartella, indicare anche una notazione puntata, come avviene per i package. In questo caso si può usare it.ioprogrammo.jspAdvanced.i18n.MyJsfMessages e si può mettere il file nella cartella WEB-INF/classes/it/ioprogrammo/jspAdvanced/i18n/.
Ma, in concreto, quali messaggi saranno contenuti nei file? Dipende da quali messaggi sono riferiti dall’applicazione (al momento nessuno!). Creiamo due messaggi esterni: il titolo e un saluto; il primo è il

titolo della pagina, compreso un numero che vorremmo variabile (si riferisce alla numerazione degli esempi); il secondo è un semplice messaggio di benvenuto (slo testo statico):

```
titolo=Esempio JSF num. {0}
saluto=Benvenuto in JSP uso avanzato
```

Si noti la prima riga: ogni messaggio può avere un numero qualsiasi di parti variabili; esse sono identificate da numeri (progressivi a partire da zero) racchiusi tra parentesi graffe. È compito di chi mostra il messaggio fornire il numero corretto di parametri. Ovviamente vanno completati anche i file per la lingua inglese e spagnola. Ecco come può essere scritta la pagina JSF che fa uso dei messaggi appena inseriti:

```
<f:loadBundle basename=
 "it.ioprogrammo.jspAdvanced.i18n.MyJsfMessages"
 var="msg"/>
<html>
<f:view>
<head>
 <title>
   <h:outputFormat value="#{msg.titolo}">
       <f:param value="06" />
   </h:outputFormat>
 </title>
</head>

<body>
 <h1><h:outputText value="#{msg.saluto}" /></h1>

</body>
</f:view>
</html>
```

Inizialmente viene indicato qual è il resource boundle da usare. Inoltre per poter usare dei parametri variabili non è possibile usare il semplice tag outputText, ma va usato il più sofisticato outputFormat. In Figura 2.7 le tre pagine mostrate accedendo con le impostazioni di lingua del browser, rispettivamente, italiano, spagnolo e inglese.

**Attenzione**
Per eseguire le prove con lingue diverse, settare sul proprio browser la lingua in uso. Con Firefox lo si fa da Strumenti > Opzioni > Avanzate > Scegli lingue. Con Explorer da Strumenti > Opzioni Internet > Generale > Lingue.

**Figura 2.7:** Le tre pagine con i messaggi personalizzati a seconda della lingua del browser.

## 2.5.7 Oltre l'esempio...

Nell'esempio si è fatto uso di un numero minimo di componenti; anche per i valori di configurazione sono state mostrate solo alcune

delle molte scelte disponibili per la tecnologia JSF. Si rimanda alle risorse già presentate
In Tabella 2.1 per eventuali approfondimenti. Per chi volesse approfondire la problematica i18, in Tabella 2.3 ulteriori risorse presenti sul Web.

| Sito Web | Descrizione |
|---|---|
| java.sun.com/docs/books/tutorial/i18n/index.html | Tutorial della Sun |
| www.concentric.net/~Rtgillam/pubs/JavaI18NTutorial.ppt | Developing Global Applications in Java |
| http://www.cn-java.com/download/data/book/i18n.pdf | Tutorial "Java internationalization basics" |
| http://www.javaportal.it/rw/24575/editorial.html | Articolo (in italiano) sull'internazionalizzazione delle applicazioni Java |
| http://www.onjava.com/pub/a/onjava/excerpt/javaexIAN3_chap8/index.html | Un capitolo del libro "Java Examples in a Nutshell"di D. Flanagan (quello dedicato a i18n) |
| http://developers.sun.com/prodtech/javatools/jscreator/learning/tutorials/2/demonstrating_i18n.html | Tutorial "Creating an Internationalized Application" usando Java Studio Creator 2 |
| http://developers.sun.com/prodtech/javatools/jscreator/learning/tutorials/2/internationalizingapps.html | "Understanding Internationalization", anch'esso usando Java Studio Creator 2 |

**Tabella 2.3:** Siti Web per approfondire le problematiche i18n.

# 2.6 ESTENSIONI ALLE JSF

Le JavaServerFaces offrono dei componenti standard, ma nulla vieta di estendere tali componenti con proprie librerie o personalizzando i tag esistenti. Il progetto MyFaces di Apache, per esempio, offre un certo numero di estensioni utili per risolvere numerosi problemi

che si incontrano utilizzando estensivamente le JSF. La home del progetto è *http://myfaces.apache.org/*. Oracle, uno dei membri più attivi per le JSF, ha creato un'ulteriore serie di componenti, le ADF (*http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/index.html*). Tra le caratteristiche di sicuro interesse delle ADF è di possedere il rendering per alcuni dispositivi mobili. Anche tale libreria entrerà a far parte delle librerie MyFaces (essendo stata donata da Oracle alla Apache Software Foundation).

## 2.7 IDE CON SUPPORTO PER LE JSF

Come accennato inizialmente la vera forza delle JSF è quella di poter essere gestite in maniera visuale da IDE evoluti. Al momento non tutti gli IDE permettono questo tipo di creazione visuale, probabilmente per la relativa "giovinezza" della tecnologia e per la sua (ancora) scarsa diffusione, ma alcuni lo fanno già in maniera egregia. Tra i più completi JDeveloper 10g della Oracle (pagina di riferimento *http://www.oracle.com/technology/products/jdev/index.html* ), che ha un supporto nativo anche per le librerie ADF. Non poteva mancare uno strumento della Sun: Java Studio Creator; esso è disponibile per il download alla pagina *http://developers.sun.com/prodtech/javatools/jscreator/downloads/*.

Alla pagina *http://developers.sun.com /prodtech/javatools /jscreator/learning/bookshelf/* è possibile accedere ai capitoli del libro "Java Studio Creator Field Guide" (libro ancora in sviluppo e aggiornato alla versione 2 di Java Studio Creator). Che dire di Eclipse? Purtroppo il supporto per le JSF è limito. Al momento di scrivere è disponibile un plug-in standard (WTP, Web Tools Platform project, *http://www.eclipse.org/webtools/*) ma la cui versione 1.5 (quella per cui è previsto un supporto per le JSF) tarda ancora ad arrivare. Non solo: al momento non è in programma la scrittura di un'interfaccia completamente visuale per la loro gestione (come avviene per gli altri IDE citati).

# UTILIZZARE L'ARCHITETTURA JMX

Le Java Managment eXtensions (d'ora in poi JMX) sono delle specifiche, rilasciate dalla Sun, che permettono la gestione completa di un'applicazione; tale gestione si riferisce sia alle problematiche di configurazione che a quelle di monitoraggio negli ambienti di sviluppo e/o produzione. Si vedrà come, in concreto, sia possibile far riferimento a tale tecnologia per gestire sia dei parametri di settaggio dell'applicazione(quali, per esempio, il livello di log, il numero di risorse da allocare e così via) sia di monitorarne l'andamento (come, per esempio, il numero e l'identificazione delle invocazioni alle singole pagine web).

## 3.1 PERCHÉ NASCE JMX

Le JMX sono, come idea generale, la realizzazione di un resource management. Questo si applica tanto a risorse software (applicazioni e servizi) quanto a quelle hardware. Di per sé il resource management non è cosa nuova, ma ciascuna risorsa, sia essa hardware che software, ha adottato strumenti di gestione specifici (e pertanto non standard). Questa specificità ha comportato una sostanziale incomunicabilità tra risorse e ha costretto i loro gestori ad affidarsi di volta in volta a soluzioni sviluppate ad-hoc. La tecnologia JMX si pone come livello infrastrutturale per fornire un accesso comune alle risorse e, allo stesso tempo, propone applicazioni di gestione standard e multi-canale (ovvero utilizzabili via Web secondo i protocolli http/https, ma anche utilizzando sistemi di messaggistica e così via).
L'uso della tecnologia JMX non è particolarmente invasivo nelle applicazioni, anche se è bene prevederne l'uso il prima possibile in maniera da orientare di conseguenza lo sviluppo delle stesse. Gran parte della complessità è "mascherata" dall'uso di componenti infrastrutturali standard e dalla presenza di semplici regole con cui esporre certe funzionalità, come la configurazione e il monitoring, dell'applicazione. In particolare, le JMX hanno un'architettura a tre livelli. Il livello più basso, a carico di chi sviluppa l'applicazione, è composto dagli oggetti che l'ap-

plicazione mette a disposizione per essere gestiti (MBean; contrazione per "Managed Bean", che non sono altro che JavaBean con metodi di accesso agli attributi più l'implementazione di un'apposita interfaccia). Le JMX non sono un prodotto, ma una specifica. La reference implementation proposta da Sun (integrata nel JSE5.0) è reperibile all'url http://java.sun.com/products/JavaManagement /download.html; esistono però anche implementazioni di terze parti, alcune commerciali, altre Open Source (tra i progetti più evoluti si segnala MX4J, scaricabile dalla pagina *http://mx4j.sourceforge.net*); ogni framework è, in linea di principio, "intercambiabile", purché soddisfi alle specifiche rilasciate da Sun. Per approfondire la tecnologia si può partire dalla pagina Web di riferimento, presente all'indirizzo *http://java.sun.com/products/JavaManagement*, altri interessanti documenti possono essere reperiti dalle pagine *http://www.onjava.com /pub/st/42*, *http://admc.com/blaine/howtos/jmx* e *http://www.research.ibm .com /journal/sj/401/kreger.html*. Il successo della tecnologia ha coinvolto anche tutti i principali progetti software Open Source. Basti pensare che lo stesso Tomcat può essere gestito via JMX, tant'è che JBoss usa, al suo interno, sia Tomcat sia gli altri componenti software grazie a JMX.

## 3.2 L'ARCHITETTURA JMX

L'architettura è formata da diversi livelli logici, ciascuno responsabile di un determinato servizio: Instrumentation, Agent e Remote Management (**Figura 3.1**).

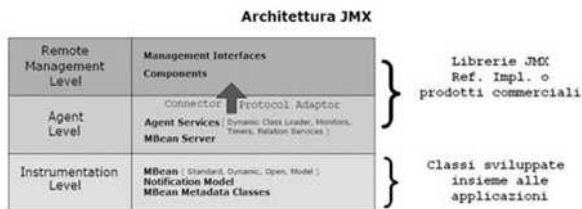



**Figura 3.1:** I livelli logici dell'architettura JMX.

Al primo livello (Instrumentation) ci sono degli oggetti, chiamati MBean, che si occupano di monitorare (e/o configurare) applicazioni e servizi (ma anche dispositivi); essi definiscono opportune proprietà e metodi per intervenire sulle risorse o conoscerne lo stato. Esistono quattro tipi di MBean (Standard, Dynamic, Model e Open); di fatto un MBean di tipo standard non differisce da una generica classe JavaBean a meno di una semplice convenzione: esso si compone di un'interfaccia che ha un nome seguito da un suffisso "MBean" e da una classe che realizza tale interfaccia (e ha il suo stesso nome a meno del suffisso).
Al secondo livello (Agent) ci sono due oggetti principali: un MBean Server (che si preoccupa di far "vivere" oggetti di tipo MBean, offrendo un opportuno ambiente) e (almeno) un agent JMX responsabile di registrare gli MBean di interesse sul server (di agent ne possono esistere anche più di uno).
Infine, nel terzo e ultimo livello (Remote Managment) ci sono adattatori (adaptors) e connettori (connectors): entrambi espongono gli Agent definiti dal livello precedente fuori dalla JVM ove questi ultimi sono stati creati e sono in esecuzione. I connettori espongono le funzionalità verso altri "mondi" JMX, mentre gli adattatori offrono supporto per client che usano protocolli standard (quali HTML, SMTP e così via).
Tutto ciò sembra complesso? In realtà non lo è affatto! Un esempio chiarirà i concetti e mostrerà quant'è semplice utilizzare quest'architettura!

## 3.3 MONITORARE SESSIONI ED ERRORI

Si supponga di voler tenere sotto controllo il numero di sessioni di un'applicazione Web. Un modo è quello di creare un bean (usando JMX si tratta di creare un MBean!) che conti sia il numero di sessioni attuali che il numero di sessioni create dall'ultimo start-up dell'ap-

plicazione; tali numeri possono essere memorizzati in due proprietà (di tipo long, per esempio). Queste proprietà vanno gestite, incrementandole entrambe quando viene creata una nuova sessione e decrementando il solo numero delle sessioni attive quando una sessione viene distrutta.

### 3.3.1 Creare l'MBean con il numero di sessioni

Un Mbean è un JavaBean come tutti gli altri ma, come si è visto, con suffisso MBean:

```
public interface SessioniMBean {
  public long getActiveSessions();
  public void setActiveSessions(long num);
  public long getTotalSessions();
  public void setTotalSessions(long num);
  public void addSession();
  public void delSession();
}
```



Si possono notare due proprietà (con i rispettivi metodi "get" e "set") e due operazioni (addSession e delSession). Ecco la classe, chiamata come la precedente ma senza suffisso MBean, che implementa l'interfaccia:

```
public class Sessioni implements SessioniMBean{
  private static long activeSessions = 0;
  private static long totalSessions = 0;

  public long getActiveSessions() {
   return activeSessions;
  }
```

```
 public void setActiveSessions(long num) {
  activeSessions = num;
 }

 public long getTotalSessions() {
  return totalSessions;
 }

 public void setTotalSessions(long num) {
  totalSessions = num;
 }

 public void addSession() {
  activeSessions++;
  totalSessions++;
 }

 public void delSession() {
  activeSessions--;
 }
}
```

## 3.3.2 Listener per le sessioni

Per conoscere quando una sessione viene creata o distrutta è necessario realizzare una classe che implementi l'interfaccia HttpSessionListener. In particolare è necessario implementare i metodi sessionCreated e sessionDestroyed (invocati, rispettivamente, quando una sessione viene creata e ogniqualvolta una sessione viene distrutta):

```
public class MySessionListener implements HttpSessionListener {
 Sessioni sessioni = new Sessioni();
 public void sessionCreated(HttpSessionEvent sessionEvent) {
```

```
 sessioni.addSession();
}

public void sessionDestroyed(HttpSessionEvent sessionEvent) {
 sessioni.delSession();
}
}
```

Tale classe deve essere registrata nel file WEB-INF/web.xml dell'applicazione, affinché, effettivamente, i suoi metodi possano essere invocati al verificarsi degli eventi opportuni:

```
<listener>
  <listener-class>
   it.ioprogrammo.jspAdvanced.MySessionListener
  </listener-class>
</listener>
```



## 3.3.3 Altri aspetti da monitorare?

In una applicazione Web ci possono essere numerosissimi altri aspetti da monitorare. Un esempio può essere il voler tracciare tutti gli errori applicativi, per esempio errori sul server (quelli con codice http 500), ma anche pagine non trovate (errori con codice http 404). Ecco un possibile MBean per gestire tali errori:

```
public interface ErroriMBean {
 public long getErroriTotali();
 public long getErrori404();
 public long getErrori500();
 public void addErrore404();
 public void addErrore500();
}
```

e la relativa implementazione:

```
public class Errori implements ErroriMBean{
 private static long errori404;
 private static long errori500;

 public long getErrori404() {
  return errori404;
 }

 public long getErrori500() {
  return errori500;
 }

 public long getErroriTotali() {
  return errori404+errori500;
 }

 public void addErrore404() {
  errori404++;
}

 public void addErrore500() {
  errori500++;
 }
}
```

I metodi del bean dovranno essere invocati dalle pagine deputate alla gestione dei due tipi di errore: infatti è possibile definire, sempre nel file WEB-INF/web.xml, quali pagine invocare in presenza degli errori 500 e 404:

```
<error-page>
```

```
  <error-code>404</error-code>
  <location>/error404.jsp</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/error500.jsp</location>
</error-page>
```

Le pagine inserite saranno pagine jsp al cui interno si invocherà il metodo addErroreXXX (per XXX si intende il numero specifico al tipo di errore che essa intercetta).

## 3.3.4 Ottenere un MBeanServer e creare un Agent

Gli MBean da soli sono davvero poca cosa. Ci vuole un server MBean per poterli gestire. A partire da Java 1.5 c'è a disposizione un tale server predefinito; per ottenerlo basta invocare:



```
ManagementFactory.getPlatformMBeanServer();
```

Ecco una possibile servlet che si preoccupa di reperire questo server e di registrarvi gli MBean appena creati:

```
package it.ioprogrammo.jspAdvanced.jmx;

public class AgentServlet extends HttpServlet implements Servlet {

  private MBeanServer server = null;

  public void init(ServletConfig config) throws ServletException {
      super.init(config);
      server = ManagementFactory.getPlatformMBeanServer();
```

```
        try {
            Sessioni sessioni = new Sessioni();
            server.registerMBean(
                sessioni,
                new ObjectName(
                 "AgentServlet:type=Sessioni")
            );
        } catch(Exception e) {
            e.printStackTrace();
        }
        try {
             Errori errori = new Errori();
             server.registerMBean(
                 errori,
                 new ObjectName(
                      "AgentServlet:type=Errori")
             );
         } catch(Exception e) {
             e.printStackTrace();
         }
 }
}
```

Ovviamente tale servlet o va invocata esplicitamente o, semplicemente, fatta partire allo start-up dell'applicazione stessa; per farlo basterà aggiungere i seguenti tag nel file web.xml (si noti il valore a "1" sul tag load-on-startup):

```
<servlet>
 <servlet-name>JMX Servlet</servlet-name>
 <servlet-class>
  it.ioprogrammo.jspAdvanced.jmx.AgentServlet
 </servlet-class>
```

```
  <load-on-startup>1</load-on-startup>
</servlet>
```

### 3.3.5 Configurare Tomcat

Affinché Tomcat esponga i propri MBean per la gestione remota è necessario aggiungere, allo script di lancio, le seguenti direttive:

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port="9004"
-Dcom.sun.management.jmxremote.authenticate="false"
-Dcom.sun.management.jmxremote.ssl="false"
```

Nel caso si sia installato Tomcat come servizio di Windows, appare sulla barra delle applicazioni un'icona che rappresenta Apache Tomcat: facendovi clic con il pulsante destro si scelga la voce "Configure...", nella finestra che appare scegliere la scheda "Java" e inserire le direttive sopracitate nella finestra "Java Options" (Figura 3.2).

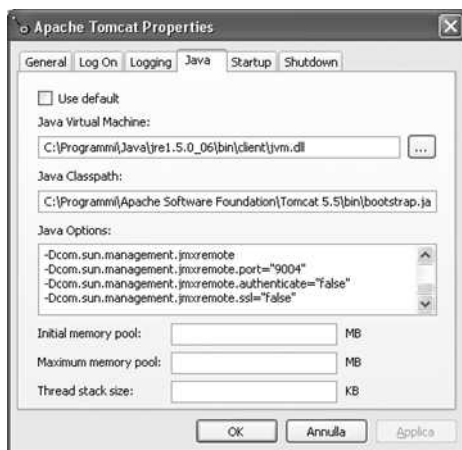



**Figura 3.2:** Configurazione di Tomcat

Ora tutto è pronto ed è possibile far eseguire Tomcat: esso partirà ed esporrà gli Mbean registrati (sia dalla nostra webapp che da Tomcat stesso ed, eventualmente, da altre applicazioni).

## 3.3.6 JConsole

Per accedere alle informazioni esposte da Tomcat si può utilizzare un qualsiasi tool JMX. Con Java 1.5 viene messo a disposizione il tool JConsole (il file si trova nella cartella bin/ dov'è installato il JDK). Eseguendo il programma appare una finestra che chiede di connettersi ad un agent; su tale finestra, nella scheda "Remote" scegliere la porta 9004 (o altra porta configurata in precedenza su Tomcat) e come url lasciare "localhost" (Figura 3.3) e quindi premere sul pulsante "Connect".

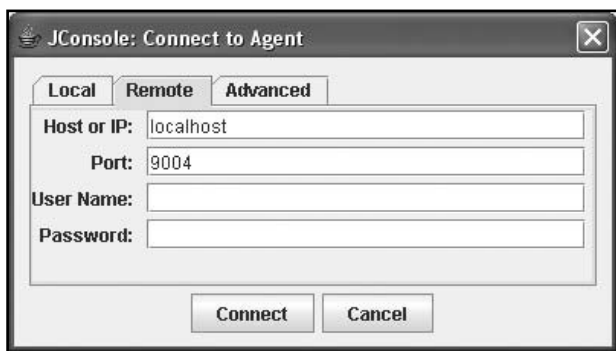



**Figura 3.3:** Esecuzione di JConsole.

Se tutto è andato a buon fine, la connessione viene aperta e vengono proposte diverse schede, ciascuna con informazioni specifiche (gestione della memoria, dei thread, delle classi, Mbean esposti e Java Virtual Machine). La sezione di nostro interesse è Mbean: una volta scelta compare la lista di tutti gli Mbean esposti. Tra essi quelli registrati con "Catalina" sono specifici di Tomcat. Quelli registrati dall'applicazione di esempio sono sotto AgentServlet (è questo il nome che gli si

è dato istanziando oggetti di tipo ObjectName): espandendo tale ramo ecco apparire i due Mbean registrati. Cliccando su uno di essi appare una finestra dov'è possibile visualizzare i valori delle proprietà, le operazioni esposte e le informazioni associate (Figura 3.4).

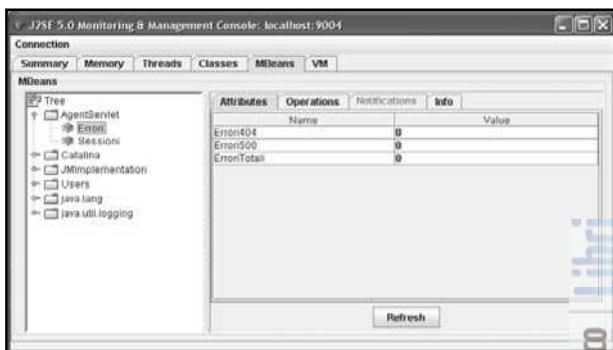



**Figura 3.4:** Mbean esposti dalla webapp.

Eseguendo diverse pagine da diverse istanze di browser si noterà che le sessioni crescono. In maniera simile, al verificarsi di errori si vedrà aumentare il numero di errori dall'MBean che ne tiene traccia.



**Attenzione**
Si noti come i valori sull'MBean Sessioni siano di colore blu, quelli di Errori siano di colore nero: questo perché i primi sono sia in lettura che in scrittura, gli ultimi solo in lettura (infatti Sessioni ha sia i metodi get che quelli set delle proprietà, Errori ha solo i metodi get!).

## 3.3.7 Gestire Tomcat con JMX

Monitorando Tomcat con JConsole si è visto che esistono numerosi MBean registrati sotto il ramo "Catalina". Curiosando su tale ramo ci si rende conto dell'enorme numero di bean esposti da Tomcat. Non deve stupire che attraverso tali bean è possibile configurare e gestire l'intero

ciclo di vita di Tomcat e delle sue applicazioni, nonché di configurare ogni suo aspetto. Un esempio fra tutti: selezionando Catalina > Web-Module, si ha accesso a tutte le applicazioni installate. Per esempio, selezionando //localhost/JspAspettiAvanzati (la webapp di esempio del libro), si arriva ad un MBean che ha, tra le sue operazioni, tutti i metodi esposti nella pagina “Tomcat Manager”! Infatti è possibile eseguire lo stop dell’applicazione, lo start o il restart (più tutti gli altri stati: init, destroy...) come evidenziato in Figura 3.5.

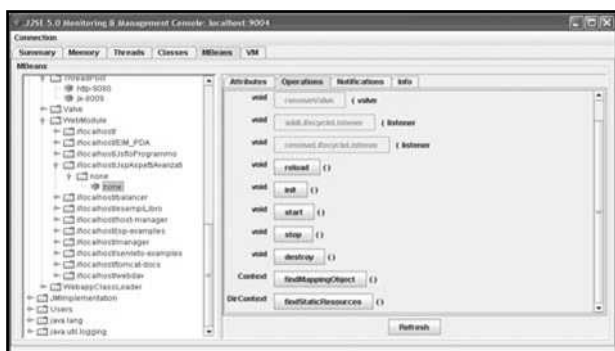



**Figura 3.5:** Tomcat espone tutte le sue funzionalità e i suoi parametri di configurazione via JMX.

## 3.3.8 JMX e altri prodotti

JMX sta diventando la tecnologia standard per la gestione/configurazione di tutti i progetti software scritti in Java.
A titolo di esempio si pensi che la quasi totalità degli application server sono gestibili attraverso JMX (per esempio lo è Sun ONE Portal Server, Oracle Application Server, JBoss, WebSphere Application Server della IBM, WebLogic della BEA).
Una lista aggiornata dei prodotti che adottano JMX è presente alla pagina *http://java.sun.com/products/JavaManagement/jmxadoption.html*.

# ACCEDERE AI DB

Accedere alle basi di dati è una problematica che va affrontata e gestita nella quasi totalità delle applicazioni Web. Utilizzare le primitive JDBC di Java è, in linea di principio, sempre possibile ma, di fatto, non viene quasi mai fatto. Il motivo è che tale tipo di primitive sono troppo di basso livello e lasciano aperti numerosi problemi (per esempio l'indipendenza dal tipo di database, benché teoricamente è garantita dai driver JDBC, in realtà viene meno quando si deve ricorrere a comandi SQL specifici, quali tipi particolari di join, funzioni di conversione di dati, date e così via). Per questo motivo sono nati dei framework che astraggono ulteriormente lo strato JDBC e forniscono funzionalità avanzate (pool di connessioni, mapping automatici o semi-automatici tra basi di dati relazionali e oggetti Java, reale dipendenza dal tipo di database e molto altro). Saranno presentati i framework open source più diffusi e, in conclusione del capitolo, si mostrerà l'uso dei data source di Tomcat.

## 4.1 QUALE DATABASE

Per effettuare delle prove si può scegliere liberamente un qualunque database, purché possieda dei driver JDBC. Negli esempi proposti si è scelto di utilizzare MySql: un prodotto open source e ampiamente usato per gestire siti Web, non solo legati al mondo Java (si può dire che è il database di riferimento per tutti i siti Web realizzati con tecnologia PHP!). Per maggiori informazioni su MySql si può far riferimento alla pagina *http://www.mysql.com*. Inoltre, se qualcuno è interessato ad utilizzare uno strumento di amministrazione accessibile dal Web, può trovare utile il progetto phpMyAdmin (pagina di riferimento *http://www.phpmyadmin.net*). Se si decide di installare anche Apache può valere la pena di semplificare l'installazione di tutti questi prodotti (MySql, PHP, phpMyAdmin e Apache) ricorrendo all'installazione di un unico software: WampServer (pagina di riferimento *http://www.wampserver.com*). Se, invece, si vuole optare per qualche altro database, non c'è che l'imbarazzo della scelta, sia in ambito Open Source (per

esempio l'ottimo Postgres) sia usando prodotti commerciali ma che possiedono versioni gratuite per lo sviluppo (è il caso di Oracle 10g e di SQL Server). Si sconsiglia l'uso di Access (benché molto diffuso per uso personale è del tutto inadeguato alla gestione di un sito Web).

## 4.2 FRAMEWORK ORM

Con il termine ORM si intende Object-Relational Mapping; un framework ORM permette di gestire classi Java che eseguono il mapping di una base dati relazionale (di solito forniscono anche strumenti avanzati sia per la creazione automatica delle classi a partire da basi di dati esistenti che, viceversa, la creazione di basi di dati a partire da un opportuno modello ad oggetti). Di framewrok ORM ce ne sono davvero molti (si veda, per una tabella comparativa tra alcune decine di essi, la pagina http://c2.com/cgi-bin/wiki?ObjectRelationalMapping). Qui vengono illustrate le principali funzioni di alcuni dei framework più usati ed interessanti tra quelli Open Source.



### 4.2.1 iBatis

Ibatis è un progetto della Apache Software Foundation che fa della semplicità il suo punto di forza; l'home page del progetto è *http://ibatis.apache.org* (**Figura 4.1**).

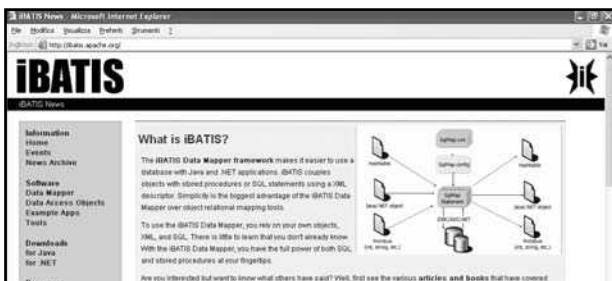



**Figura 4.1:** home page del progetto iBatis.

iBatis utilizza descrittori XML per eseguire un mapping tra JavaBean (e altre entità applicative) con istruzioni SQL. Le istruzioni SQL sono istanziate, di volta in volta, con i valori provenienti dall'oggetto con cui è stato fatto il mapping: in questo modo sia gli attributi di un JavaBean che oggetti di eventuali strutture dati (come Map o altri oggetti Java) sono considerati come istanze per eseguire select, inserimenti e altre istruzioni SQL. Il risultato dell'istruzione, a sua volta, viene mappata o con oggetti JavaBean o con opportune collezioni di dati.
Per semplificare la creazione dei file di mapping e le classi Java (sia JavaBean che rappresentano tabelle relazionali che oggetti DAO) è stato creato un tool: Abator (*http://ibatis.apache.org/abator.html*). Benché il tool non sia ancora considerato stabile, è un ottimo punto di partenza per generare gli oggetti necessari ad iBatis per interagire con una base dati esistente. Abator può essere eseguito sia come programma stand-alone che come plug-in di Eclipse.

## 4.2.2 Hibernate

Hibernate è, con tutta probabilità, il framework ORM più utilizzato in assoluto. La pagina di riferimento del progetto è http://www.hibernate.org (**Figura 4.2**). Questa sua diffusione è dovuta sia alla stabilità del pro-

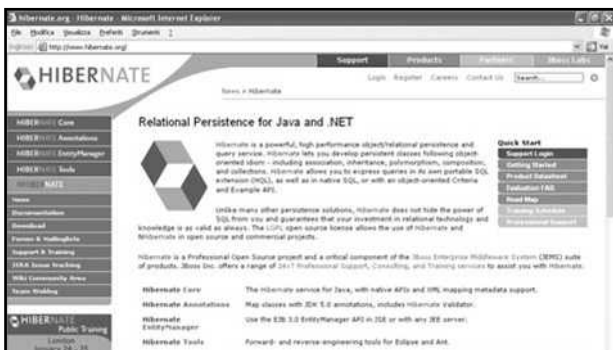



**Figura 4.2:** home page del framework Hibernate.

dotto, al suo tipo di licenza Open Source e alla relativa semplicità d'uso. La riprova della bontà del progetto è rappresentata dal fatto che, oltre a un numero consistente di applicazioni Java che ne fanno uso, il framework è stato "migrato" anche in ambiente .NET: infatti NHibernate ne ricalca la filosofia del framework (originariamente creato per Java) per la piattaforma .NET. Da non molto il progetto è stato incluso nei progetti di JBoss (con evidente miglioramento in termini di supporto e documentazione) e può contare anche su un interessante plug-in per Eclipse per poter generare il reverse engineering di basi dati esistenti (creando in maniera semplice i file XML per il mapping). Tale plug-in è scaricabile dalla pagina http://www.hibernate.org/255.html. Sul web sono disponibili numerosi esempi e tutorial su come integrare Hibernate con i più diffusi framework MVC e altre tecnologie Java. In **Tabella 4.1** alcuni esempi.

| Pagina | Descrizione |
|---|---|
| http://www.javaworld.com/javaworld/jw-01-2005/jw-0124-strutshibernate.html | "Struts Recipes: Hibernate and Struts"; integrazione con il framework MVC Struts |
| http://www.opensymphony.com/webwork/wikidocs/Hibernate.html | Mostra come integrare Hibernate in WebWork (risorsa reperita dal wiki di quest'ultimo) |
| http://www.gloegl.de/5.html | "Hibernate Tutorial - The Road to Hibernate" |
| http://www.developer.com/java/ent/article.php/10933_3577101_1 | "Developing J2EE Applications Using Hibernate Annotations and Spring MVC" |
| http://www.onjava.com/pub/a/onjava/2005/05/18/swingxactions.html | "Wire Hibernate Transactions in Spring" |
| http://jakarta.apache.org/turbine/turbine/turbine-2.4-M1/howto/hibernate-howto.html | HOW-To su come integrare Hibernate usando il framework Turbine |
| http://wiki.apache.org/myfaces/Hibernate_And_MyFaces | "Integration of Hibernate and MyFaces" |
| http://blog.exadel.com/?p=8 | Use Hibernate and JSF with Less Coding |

**Tabella 4.1:** Tutorial per integrare Hibernate con altri framework e tecnologie Java.

## 4.2.3 Cayenne

Cayenne è un prodotto open source sviluppato dall'azienda ObjectStyle (pagina di riferimento http://www.objectstyle.org/cayenne/). Dal Marzo 2006 è entrato a far parte dei progetti della Apache Software Foundation (al momento della stesura del libro è ancora un progetto nella fase "incubator": in pratica deve dimostrare di essere un prodotto sufficientemente maturo e, allo stesso tempo, avere una comunity sufficientemente ampia e stabile che ne segue lo sviluppo anche se si dovessero verificare alcune defezioni del team di sviluppo originario; terminata questa fase farà parte a pieno titolo dei progetti "sponsorizzati" da Apache).

**Figura 4.3:** home page attuale del progetto Cayenne (a breve la home sarà sul sito di Apache).

Il progetto gode di ottime caratteristiche sia per la semplicità d'uso che per le performance ottenibili. Tra le altre cose permette l'uso di una cache a diversi livelli e di un tool grafico sia per il reverse engineering di una base esistente che per la creazione della stessa a partire da un modello di classi Java. Ottima anche la documentazione che accompagna il progetto, a cui si rimanda per i dettagli d'uso.

## 4.2.4 Castor

Castor (http://castor.codehaus.org/) rappresenta un tool di mapping tra oggetti Java, documenti XML e tabelle relazionali. In questo contesto la parte che ci interessa è il mapping tra oggetti Java e tabelle relazionali (Castor JDO). Il framework offre un layer di persistenza per oggetti Java basandosi su schemi di mapping definiti dall'utente. Offre supporto per le transazioni (in pratica Castor gestisce automaticamente i lock sugli oggetti trattati; l'utente può definire diversi meccanismi di locking, personalizzandoli per l'applicazione). Definisce un linguaggio, OQL (Object Query Language Specification) per interagire con il database in maniera indipendente dal dialetto SQL adottato. Infatti tale linguaggio permette di invocare metodi su oggetti Java anziché invocare istruzioni SQL specifiche per il database in uso. Permette l'uso di cache per minimizzare l'accesso al database e gestisce eventuali collezioni associate ad un oggetto con la lazy loading (in pratica gli oggetti in relazione con quello principale vengono letti solo quando si accede ad essi, minimizzando le interazioni con la base dati a quelle indispensabili).

# 4.3 TROPPI FRAMEWORK? POJO!

Siccome nessun framework ha il sopravvento sull'altro, il risultato è che la scelta di uno rispetto all'altro è lasciata, nel migliore dei casi, allo sviluppatore, nel peggiore al cliente (è il caso in cui il cliente è una grossa azienda o istituzione con proprie regole aziendali per lo sviluppo; per uniformità può richiedere di utilizzare strumenti e framework specifici). Il problema è sia di dover imparare framework diversi sia di non avere un'architettura omogenea tra le diverse applicazioni. Tale omogeneità permetterebbe sia un facile riuso di codice fra le diverse applicazioni sia un minor tempo nell'estensione di un'applicazione quando sopraggiungono nuovi requisiti (manutenibilità del codice). Un modo per astrarre l'architettura dal framework ORM utilizzato c'è, e si chiama Plain Old Java Objects (POJO, http://www.martinfowler.com/bliki/POJO.html). In pratica

una classe è POJO se non estende alcun altra classe né implementa interfacce specifiche. In ultima analisi basta che sia un JavaBean senza relazioni con il framework usato per renderlo persistente. Questo minimizza l'impatto del framework stesso su tutta l'applicazione rendendo possibile, in linea di principio, la sostituzione del layer di persistenza senza intaccare minimamente il resto dell'applicazione. Ma è proprio necessario usare un framework ORM? Probabilmente sì, per lo meno usando applicazioni complesse. Altre volte basta usare un pool di connessioni. Spesso, anche usando i framework, si può demandare la gestione del pool al container dell'applicazione. Vediamo come farlo con Tomcat...

## 4.4 USARE I DATA SOURCE DI TOMCAT

Tomcat mette a disposizione un pool di connessioni gestite a livello di container. Questo è utile sia quando si vuol accedere ai database via JDBC sia quando si vuole usare un framework di più alto livello, come quelli presentati in precedenza (infatti, benché la maggior parte di essi implementi un proprio pool, essi possono far uso anche di pool esterni). Tomcat espone il pool utilizzando il meccanismo JNDI (Java Naming and Directory Interface) in maniera compatibile con le specifiche J2EE.

**Attenzione**
Tomcat fa uso di una libreria che offre un pool di connessioni al container; tale libreria si chiama DBCP e si basa su Jakarta-Commons Database Connection Pool (jakarta.apache.org/commons/dbcp/); i "commons" sono packages di funzionalità base per il progetto Jakarta.

### 4.4.1 Creare la base dati

Ovviamente, per poter leggere dei dati è necessario creare un database (con un utente abilitato ad esso) e almeno una tabella. Ecco le istru-

zioni da eseguire (per eseguirle o si usa la console testuale fornita insieme a MySql o una console grafica come quella di phpMyAdmin):

```
> GRANT ALL PRIVILEGES ON * . * TO 'ivan'@'%'
 IDENTIFIED BY 'venuti' WITH GRANT
 OPTION MAX_QUERIES_PER_HOUR 0
 MAX_CONNECTIONS_PER_HOUR 0
 MAX_UPDATES_PER_HOUR 0 ;
```

Con questa istruzione si è definito l'utente 'ivan' con password 'venuti'. Ora si crei un database chiamato 'jspdb':

```
> CREATE DATABASE `jspdb` ;
```

Infine ecco una tabella in cui si memorizzeranno dei log di accesso (dati da memorizzare: un id, chiave auto-incrementante, la data di accesso e l'ip con cui l'utente si è connesso):

```
> create table logger (
    id int not null auto_increment primary key,
    quando date,
    ip varchar(255) );
```

## 4.4.2 Definire un contesto per la webapp

Una web application può aver definito un proprio contesto nel file /conf/server.xml; tale contesto è racchiuso tra i tag <Context> e </Context> (nella distribuzione standard di Tomcat ci sono già dei contesti di esempio predefiniti). Ecco come potremmo creare un contesto per la nostra applicazione con, al suo interno, una risorsa (specifica ad essa e non visibile dalle altre applicazioni):

```
<Context path="/JspAspettiAvanzati" docBase="JspAspettiAvanzati"
```

```
    debug="5" reloadable="true" crossContext="true">
 <Resource name="jdbc/TomcatDbPool"
   auth="Container" type="javax.sql.DataSource"
   maxActive="100" maxIdle="30" maxWait="10000"
   username="ivan" password="venuti"
   driverClassName="com.mysql.jdbc.Driver"
   url="jdbc:mysql://localhost:3306/jspdb?autoReconnect=true"
   removeAbandoned="true"
   removeAbandonedTimeout="60"
   logAbandoned="true"
  />
</Context>
```

La risorsa appena definita, il cui nome è "jdbc/TomcatDbPool", possiede numerosi parametri di configurazione. In Tabella 4.2 il loro significato.

| Parametro | Significato |
|---|---|
| name | Nome della risorsa |
| auth | Chi risolve il problema dell'autenticazione (in questo caso il container). |
| type | Tipo di risorsa (quale classe Java la realizza) |
| maxActive | Il numero massimo di connessioni che il pool può contenere. Se vale 0 non c'è limite; attenzione a far sì che il database MySql gestisca un numero superiore di connessioni specificate in questa proprietà! |
| maxIdle | Numero massimo di connessioni da mantenere nel pool quando hanno stato idle. Se vale -1 non c'è limite. |
| maxWait | Numero di millisecondi di attesa prima di lanciare una eccezione in caso di mancato collegamento al db. -1 significa aspettare indefinitivamente. |
| username | Nome dell'utente con cui accedere al db |

| password | Nome della password (alcuni driver non permettono di specificare password vuote!). |
|---|---|
| driverClassName | Nome della classe del driver JDBC. |
| url | Stringa JDBC di connessione al database. Essa dipende dal tipo di database in uso. Nel caso di MySql il valore autoReconnect=true permette di riconnettersi alla base dati quando la connessione viene chiusa dal server db. |
| removeAbandoned | Questo, insieme ai parametri che seguono, permette di risolvere il problema di applicazioni non scritte in maniera appropriata e che lasciano aperte delle connessioni. Così facendo non permettono il loro riuso (cosa che può portare ad un blocco dell'applicazione e del db stesso!). Impostando questo attributo a "true" il container tenta di risolvere da sé il problema. |
| removeAbandonedTimeout | Specifica il numero di secondi che devono passare prima di considerare una connessione abbandonata. Di default vale 300 secondi. |
| logAbandoned | Quando viene settato a true questo attributo, ogni volta che viene rimossa una connessione abbandonata, viene mostrato uno stack trace in cui si evidenzia il codice che ha generato il problema (molto utile per capire dove risiede il problema e quindi dove intervenire per risolverlo). Di default vale false. |

**Tabella 4.2:** Attributi per definire un data source a livello di container (Tomcat 5.5).

## 4.4.3 Riferirsi alle risorse del contesto

Nella webapp è possibile riferirsi alle risorse definite nel suo contesto intervenendo sul file /WEB-INF/web.xml:

```
<resource-ref>
 <description>Pool di connessioni dal contesto</description>
 <res-ref-name>jdbc/TomcatDbPool</res-ref-name>
```

```
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

## 4.4.4 Accedere ai dati

Non resta che scrivere la pagina JSP per accedere alla tabella. In particolare si potrebbe, nella stessa pagina, registrare la richiesta attuale come nuovo record sul database (prendendo l'ip da cui proviene la richiesta e la data di sistema). Supponendo di utilizzare le JSTL e, in particolare, la libreria sql, ecco come si potrebbe creare il comando di inserimento che fa riferimento al data source definito dal container:

```
<sql:update var="nuovo"
  dataSource="jdbc/TomcatDbPool"
  sql="insert into logger(ip, quando) values(?, ?)">
   <sql:param value="<%= request.getRemoteAddr() %>" />
   <sql:param value="<%= new java.util.Date() %>" />
</sql:update>
```

In maniera simile è possibile definire l'istruzione che esegue la select:

```
<sql:query var="esempio"
 startRow="0" maxRows="100"
 dataSource="jdbc/TomcatDbPool"
 sql="select ip, quando from logger order by quando desc"
/>
```

mentre ecco il codice che stampa dapprima i nomi delle colonne e poi i valori reperiti sulla tabella:

```
<table border=1>
<tr bgcolor="#EEEEEE">
<c:forEach items="${esempio.columnNames}" var="nomi">
```

```
  <td>    <b>${nomi}</b>  </td>

</c:forEach>
</tr>
<c:forEach items="${esempio.rows}" var="elem">
<tr>
  <c:forEach items="${elem}" var="col">
  <td>    ${col.value}  </td>

  </c:forEach>
</tr>
</c:forEach>
</table>
```

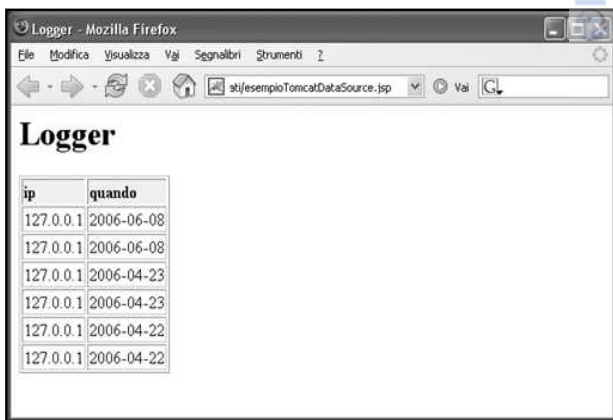
Nel file esempioTomcatDataSource.jsp si può vedere l'esempio completo: rispetto al codice riportato esegue anche un controllo sugli errori e verifica se il numero di record della tabella era maggiore del limite impostato (100) e, se sì, avvisa l'utente con un apposito messaggio (Figura 4.4).



**Figura 4.4:** l'esempio che fa uso di datasource.

**Attenzione**
È bene ricordarsi che va inserito sempre il file JAR contenente i driver JDBC per il database utilizzato. Tale JAR può essere messo nella cartella /WEB-INF/lib/ dell'applicazione, ma, più comunemente, lo si mette nella cartella $CATALINA_HOME/common/lib; in questo modo tutte le webapp installate possono condividere tale libreria.

# APPLICAZIONI WEB E SICUREZZA

Essendo le applicazioni Web esposte a chiunque, per loro natura, ci si deve sempre porre il problema della sicurezza, soprattutto quando si gestiscono informazioni sensibili o riservate. Inoltre, il più delle volte, anche applicazioni Web che offrono solo dati pubblici sono suscettibili di attacchi. Se un'applicazione non è configurata a dovere, un utente malintenzionato potrebbe corrompere i dati pubblicati o fare un uso illecito delle risorse (come utilizzare la propria applicazione per effettuare attacchi a siti di terze parti). La complessità della problematica ha fatto sorgere tutta una serie di iniziative affinché i responsabili e gli amministratori dei siti, ma anche gli sviluppatori di applicazioni, siano informati e consci dei rischi intrinseci alla tecnologia Web. Uno tra i progetti più autorevoli è Open Web Application Security Project, OWASP (il cui sito di riferimento è *http://www.owasp.org*, **Figura 5.1**) , pensato per aiutare le organizzazioni a capire il problema della sicurezza nell'ambito delle ap-

**Figura 5.1:** Il sito ufficiale di Open Web Application Security Project

plicazioni Web e a migliorarne le soluzioni.
Tra le molte iniziative e informazioni, sul sito sono reperibili anche le prime 10 criticità da affrontare (*http://www.owasp.org/index.php/OWASP_Top_Ten_Project*) elencate in ordine di importanza:

**1)** Input non validato
**2)** Forzatura del controllo per le restrizioni d'accesso
**3)** Forzatura delle politiche di autenticazione e di gestione della sessione
**4)** Cross Site Scripting (XSS)
**5)** Buffer Overflow
**6)** Code injection
**7)** Errata gestione degli errori
**8)** Algoritmi e tecniche di crittografia "deboli"
**9)** Denial of service
**10)** Configurazione e politiche di gestione dei server deboli

Si vedranno i punti direttamente sotto il controllo dell'applicazione (infatti spesso i problemi sono "a monte", non gestibili dal programmatore: è il caso del punto 10). Quando possibile verrà mostrato come questi problemi possano essere presenti anche nelle applicazioni scritte con tecnologia J2EE. Si vedrà com'è fondamentale che l'applicazione sia stata concepita in maniera da avere una buona architettura di base e da prevedere i tipi di attacco più comuni. Si descriverà per ultimo il primo problema (input non validato) perché verrà affrontato in maggior dettaglio alla fine del capitolo, insieme ai punti 4, 5, 6, in quanto fanno tutti riferimento allo stesso problema: i dai provenienti dall'utente.

## 5.1 FORZATURA DEL CONTROLLO PER LE RESTRIZIONI D'ACCESSO

Si supponga di limitare l'accesso alle sole richieste provenienti da partico-

lari indirizzi IP (fidati). Questo è molto semplice da realizzare nelle Servlet e nelle pagine JSP in quanto si hanno a disposizione i seguenti metodi:

```
request.getRemoteAddr();
request.getRemoteHost();
```

Il primo restituisce l'indirizzo IP, il secondo il nome dell'Host. Si è a posto? Purtroppo no! Infatti il dato dell'indirizzo IP di chi effettua la richiesta può essere inserito in maniera arbitraria se si costruiscono le richieste http a basso livello! Pertanto questo tipo di restrizione è facilmente forzabile. Questo è solo un esempio per indicare come il programmatore debba sempre verificare se le informazioni di cui fa uso sono certe o soggette a possibili forzature.

### 5.1.1.Forzatura delle politiche di autenticazione e di gestione della sessione

Se si usano parametri passati nella query string, si sappia che essi sono reperibili da chiunque, in quanto le URL restano nella cache dei browser. Questo presenta notevoli problemi di sicurezza nel caso di uso di computer ad accesso pubblico in cui username e password vengono passati nella query string (soprattutto se, punto 8, vengono applicati algoritmi di sicurezza deboli!). Lo stesso problema si può presentare nella gestione delle sessioni quando esse sono gestite da parametri provenienti dal client (per esempio identificativi di sessione passati in form html o, di nuovo, sulla query string). Si pensi al malaugurato caso in cui si decide di utilizzare un progressivo per identificare la sessione corrente! È fin troppo semplice provare a "indovinare" un numero di sessione valido, soprattutto se, lato server, non vengono fatti ulteriori controlli quali indirizzo IP da cui proviene la richiesta o cookie presenti nel client (benché anche questi controlli siano soggetti a forzatura, è senz'altro molto più com-

plesso “indovinare” sia il numero di sessione sia quest’altra informazione!).

## 5.1.2 Errata gestione degli errori

Se si fa propagare l’errore fino alla pagina di visualizzazione, è probabile che esso contenga informazioni riguardanti l’implementazione (tecnologia utilizzata, ma anche versioni di prodotti e quando, per esempio, avvengono eccezioni accedendo alla base dati perché momentaneamente non accessibile, informazioni riservate quali password e nomi di utenti). In questi casi si parla di una errata gestione degli errori.
Si è avuto modo di vedere la direttiva page con l’attributo error, per specificare pagine di errore personalizzate all’interno delle pagine JSP:

```
<%@ page errorPage="isErrorPage.jsp" %>
```

Questo però dovrebbe essere fatto su tutte le JSP, lasciando la possibilità di “dimenticanze”. Un modo alternativo di procedere è configurare il Servlet Container per specificare il comportamento in caso di errori. Ecco come si possono specificare nel file web.xml le pagine di gestione degli errori:

```
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/isErrorPage.jsp</location>
</error-page>
```

Se, in exception type, si scrive un’eccezione di tipo particolare, è possibile utilizzare diverse pagine di errore per diversi tipi di eccezione:

```
<error-page>
  <exception-type>java.lang.NullPointerException</exception-type>
  <location>/NullErrorPage.jsp</location>
</error-page>
```

```
<error-page>
  <exception-type>java.lang.ClassCastException</exception-type>
  <location>/ClassCastErrorPage.jsp</location>
</error-page>
```

La cosa interessante (e non ottenibile utilizzando solo le JSP con le direttive page e l'attributo errorPage) è che si possono intercettare anche gli errori http. Per esempio ecco come gestire con una pagina personalizzata l'errore 404 (risorsa non presente) o altri errori http (un po' come è stato fatto negli esempi del Capitolo 3):

```
<error-page>
  <error-code>404</error-code>
  <location>/ErrorPage404.jsp</location>
</error-page>
```

Esistono poi degli errori più "subdoli"; si pensi al caso in cui si assume che l'operazione venga fatta una ed al più una sola volta (si pensi al pulsante "Conferma" di un ordine: se la pagina successiva è lenta a caricarsi l'utente potrebbe premere sullo stesso pulsante un numero imprecisato di volte!). Per questo tipo di problemi si veda l'interessante articolo "JSP Error pages and Preventing Repeated Operations", alla pagina *http://java.sun.com/developer/EJTechTips/2003/tt0114.html*.

## 5.1.3 Algoritmi e tecniche di crittografia "deboli"

Un aspetto fondamentale delle tecniche di crittografia è quello che non si può "sperare" che la forza di una tecnica stia nella segretezza di un algoritmo.
In pratica sommare un numero fisso (per esempio 2, 5, 12, …) ad ogni carattere (rispetto alla sua tabella di codifica) prima di utilizzare il da-

to è un algoritmo debole, in quanto l'unica "sicurezza" sta nella non conoscenza dell'algoritmo (infatti, conosciuto l'algoritmo, basta sottrarre lo stesso numero e si ottiene il messaggio "in chiaro"!).
Purtroppo la maggior parte delle tecniche di crittografia fatte "a mano" sono proprio deboli perché si basano su questo presupposto errato (o meglio: più che errato troppo semplice da scardinare, come l'ampia letteratura sull'argomento ha più volte dimostrato).
In Java, per fortuna, esistono packages dedicati alle tecniche di crittografia e sono tutti algoritmi robusti. Tali package sono quelli della gerarchia java.security.

## 5.1.4 Denial of service

Essendo le risorse Web accessibili ad un numero imprecisato di utenti, si è soliti dimensionare le risorse in base al numero di utenti previsti in un certo lasso di tempo. Il problema nasce sia quando questa stima si rivela errata sia quando si è in presenza di veri e propri attacchi; è in questo caso che si parla di attacchi di tipo DoS (Denial of Service).
Di solito sono attacchi perpetrati da programmi automatici o da veri e propri virus; in quest'ultimo caso la prima condizione essenziale per il successo dell'attacco è la diffusione del virus (magari sfruttando falle di sicurezza di Web Server o altri prodotti software in "ascolto" di richieste sul Web); successivamente, al verificarsi di particolari eventi (date o comandi) tutte le postazioni infette effettuano chiamate al servizio sotto attacco (in questo caso si parla anche di DDoS: Distributed Denial Of Service).
Le soluzioni a questo problema sono per lo più esterne all'applicazione (configurazione appropriata del Web Server, router e altri dispositivi di rete), ma molto può essere fatto anche a livello applicativo. Infatti è fondamentale identificare i servizi a rischio (solitamente quelli che richiedono molte risorse per portare a termine l'operazione) e proteggerli in maniera opportuna.

## 5.1.5 Problemi con i dati provenienti dall'utente

Una considerazione generale: non ci si può mai fidare dai dati provenienti dal client. Da qui la problematica della verifica di quanto arriva sul server. Di seguito si analizzeranno diversi tipi di errori che un input "malevole" può causare e si vedrà quale sia la loro fonte. Ovviamente, per ogni tipologia, sono forniti suggerimenti per eliminare (o minimizzare) la causa e i problemi derivanti da comportamenti "anomali".

## 5.1.6 Input non validato

Benché molte pagine contengano controlli sintattico/semantici lato client (esempio controlli Javascript), può essere semplice forzare tali controlli effettuando richieste http da pagine esterne. Il problema si fa ancora più delicato qualora i dati inseriti dall'utente vengano memorizzati, in maniera persistente, su una base dati.

Successivamente essi, per essere "trattati" o semplicemente visualizzati, appaiono all'interno di altre pagine Web (per gli utenti che sono abilitati alla loro gestione o come dati visibili a tutti). È necessario prestare attenzione a questi problemi:

**1.** il tipo di dato non corrisponde a quello che il database memorizza (per esempio si è digitato del testo anziché un numero oppure si è scritto un numero con delle cifre decimali quando invece dovrebbe essere intero e così via);

**1.2**. la lunghezza dei dati non è quella prevista (per esempio si tenta di memorizzare una stringa o un numero troppo grandi); è in questo caso che si possono verificare problemi di buffer overflow;

**1.3**. benché il tipo di dato inserito sia corretto, alcuni caratteri possono creare problemi sia quando si memorizzano i dati (quindi caratteri che hanno un significato particolare per il database) sia quando li si mostrano (caratteri che hanno un significato particolare per il client, che spesso è un browser Web, come i caratteri "<" e ">").

## 5.1.7 SQL Injection

Quando si  interagisce con il database bisogna fare attenzione a come lo si fa. Di seguito viene presentato un esempio di codice potenzialmente insicuro; esso è composto da una classe tra i cui metodi ce n’è uno che permette di eseguire un comando SQL passandogli la stringa del comando:

```
public int executeSql(String quale) throws Exception{
 Connection con = null;
 Statement stmt = null;
 try {
  con = this.getConnection();
  stmt = con.createStatement();
  if (stmt.execute(quale)) {
   ResultSet rs = stmt.getResultSet();
   rs.last();
   int size = rs.getRow();
   rs.close();
   return size;
  }

  return 0;
 } finally {
 // libera le risorse
 }
}
```



Potremmo scrivere una pagina JSP che chiede nome utente e password, grazie ad una form del tipo:

```
<form name="login" action="test.jsp">
 <input type="text" name="user" value="" />
 <input type="password" name="pswd" value="" />
```

```
 <input type="submit" value="Invia" />
</form>
```

Un’opportuna classe sul server, invocata da una Servlet o una JSP, verifica se l’utente è tra quelli abilitati all’uso del programma. Per verificarlo basterebbe eseguire il seguente comando:

```
int numRows = g.executeSql(
 "SELECT * FROM utenti WHERE user='"+user+
 "' AND password='"+pswd+"'");
```

dove user e pswd sono i valori letti dalla request. Purtroppo questa soluzione, benché semplice, porta alla possibilità, per chi invia i dati, di effettuare una login con successo in qualsiasi caso o, peggio ancora, di eseguire codice arbitrario sul server. Per esempio se, sul il campo contenente la password, vengono digitati i seguenti caratteri:

```
' OR '1'='1
```

Allora il comando SQL che viene eseguito è:

```
SELECT * FROM utenti WHERE
 user='quello digitato' AND
 password='' OR '1'='1'
```

In questo modo la select restituirà sempre tutti i record della tabella, in quanto l’ultima condizione è sempre vera!
Ovviamente l’utente potrebbe scrivere, sempre sul campo password, qualcosa come

```
'; INSERT INTO tabella values(1, 'test', 'test
```

Inserendo così nuove righe sulla tabelle; allo stesso modo potrebbe

eseguire operazioni di update, delete o qualunque altra operazione abilitata (vale la pena ricordare che MySQL nega, come impostazione predefinita, la possibilità di eseguire più comandi separati da ";", proprio per i motivi di sicurezza appena accennati; altri database non hanno questo tipo di protezione).
Tra i caratteri che creano problemi ci sono anche i caratteri che identificano commenti ma anche valori NULL o altre stringhe particolari. Se ci si aspetta dei numeri, è comune che essi vengano controllati solo dal client (controlli Javascript) esponendo il server allo stesso genere di attacchi descritti poc'anzi, con l'aggravante che l'utente non deve nemmeno preoccuparsi di aprire/chiudere apici, ma solo di comporre stringhe che abbiano un significato particolare come istruzione SQL.



## 5.1.8 SQL Injection: come ci si difende?

Quando possibile, è meglio evitare di utilizzare il codice SQL per eseguire istruzioni sul database, ma bisogna usare tecniche alternative. In Java, per esempio, è possibile utilizzare i PreparedStatement. Se non è possibile e si deve per forza far eseguire istruzioni SQL, è necessario prevedere un filtro sui dati prima di inserirli nel database. La scrittura di un filtro può essere fatta in due modi: filtro per una validazione positiva o filtro per una validazione negativa. Il primo prevede di specificare tutti e soli i caratteri ammessi (la validazione ha effetto "positivo", ovvero dice quali stringhe vanno bene). È possibile scrivere uno di questi filtri, per esempio, validando l'input attraverso opportune espressioni regolari; ecco un'espressione che identifica solo i caratteri che sono lettere dell'alfabeto (non accentate) e numeri:

```
s/[^a-zA-Z0-9]//\
```

Se invece non è possibile escludere a priori certi caratteri, è necessario eseguire controlli sui caratteri che possono creare problemi (per esempio eseguendo il loro "escape", ovvero verificare in che modo un carat-

tere possa essere codificato per restare un carattere valido per il database). È il caso dell'apice singolo, che spesso si deve accettare come input (pena la non validità di stringhe quali cognomi come "D'Angelo" o descrizioni come "L'esempio"). In tali casi si realizzano i filtri con validazione "negativa" (ovvero che dicono quali sono le stringhe non valide) e che prevedono di trasformare stringhe non valide in stringhe valide; in SQL è possibile usare due singoli apici per specificare che il carattere non è la terminazione (o inizio) di una stringa ma un carattere apice. Spesso è necessario creare una libreria per questo tipo di funzionalità. Ecco un esempio di metodo Java per tale scopo:

```
public String apiceSql(String old) {

 StringBuffer buf = new StringBuffer();
 for (int i = 0; i < old.length(); i++) {

   if (old.charAt(i) == '\'')
     buf.append("''");
   else
     buf.append(old.charAt(i));

 }
 return buf.toString();
}
```

Però bisogna prestare attenzione a vari fattori, come possibili alternative di codifica dei dati (per esempio la codifica Unicode prevede opportune codifiche per i caratteri ASCII standard, compreso l'apice singolo). In questo caso è possibile "offuscarli", magari utilizzando metacaratteri e, in generale, non sono sempre preventivabili tutte le insidie che si possono nascondere nel non poter "validare" l'input. Per questo motivo, quando possibile, è consigliato usare solo filtri a validazione positiva.

## 5.1.9 SQL Injection e attacchi di “secondo ordine”

È necessario far sì che non solo le applicazioni Web pubbliche filtrino la visualizzazione di dati provenienti da un utente, ma che lo facciano anche (eventuali) applicazioni gestionali o applicazioni che elaborano dati provenienti da altre applicazioni Web. Un esempio su tutti: alcune applicazioni Web potrebbero tener traccia, sul database, delle informazioni sui browser che hanno avuto accesso all’applicazione (per esempio memorizzando gli header delle richieste http). È possibile “estrarre” informazioni quali il tipo di browser utilizzato (quale produttore) ma anche il sistema operativo che lo esegue e così via. Purtroppo è semplice realizzare “camuffamenti” anche su tali valori. Pertanto si potrebbero scrivere valori arbitrari in tali header. Si supponga che vi venga scritto del codice Javascript! Se anche l’applicazione che memroizza questi valori non li visualizzasse mai, potrebbe esistere una seconda applicazione Web che ha la necessità, per fini statistici, di visualizzare le informazioni del record. Se quest’ultima non usasse particolari precauzioni, il codice Javascript verrebbe mandato in esecuzione aprendo il record sulle sue pagine Web (attacco di Cross Site Scripting)! Si parla di attacchi di secondo ordine in quanto il “danno” non è immediato, ma è latente nel sistema e si scatena al verificarsi di particolari eventi o condizioni (come la visualizzazione del contenuto dell’header in un secondo tempo). Esistono altri problemi classificabili come attacchi di secondo ordine. Per esempio si supponga che l’utente inserito sia “admin'--”; è facile prestare attenzione al fatto che tale stringa venga memorizzata in inserimento quando l’utente inserisce il dato, per esempio usando il metodo apiciSql visto in precedenza:

```
String sql = "INSERT INTO utenti VALUES " +
  "('"+apiciSql(user)+"',"
  " '"+apiciSql(pswd)+ "')";
```

Ma che accade quando l’utente modifica la password? Dipende da co-

me è stato scritto il codice: in alcuni casi potrebbero nascondersi insidie difficili da individuare; ecco un esempio:

```
stmt.execute("UPDATE utenti SET "+
  "password='"+apiciSql(pswd)+"' "+
  "where user='"+rs.getString(0)+"'");
```

Dove rs è un ResultSet contenente un risultato di una selezione precedente sul nome dell'utente. In questo caso la stringa SQL è, quando lo username è "admin'--":

```
UPDATE utenti SET password='nuova' where user='admin'--'
```

Questa stringa risulta, per molti db che interpretano i caratteri "--" come specifica che quel che segue è solo un commento; pertanto essa è equivalente a:

```
UPDATE utenti SET password='nuova' where user='admin'
```

una stringa SQL valida per la modifica della password dell'utente "admin"! È ovvio che l'errore è stato quello di non applicare il filtro negativo anche al valore reperito dal db, ma è ovvio solo perché si è mostrato il possibile effetto collaterale: molto meno ovvio è trovare errori "logici" di questo tipo quando l'applicazione è complessa ed esistono numerosi punti dove controllare le interazioni con il db…

## 5.1.10 Code Injection

Come accennato precedentemente, esiste un'altra classe di problemi, legata alla visualizzazione dei dati dovuta a code injection. Se si tratta di codice HTML l'unico "danno" potrebbe essere quello di sporcare la visualizzazione della pagina; se invece l'utente scrive del codice Javascript allora i rischi sono molto più grandi e si usa il termine di Cross Site Scrip-

ting (XSS). Supponiamo, per esempio, che un sito offra la possibilità ad ogni visitatore di lasciare un commento (guestbook). Se chi scrive il commento include dei caratteri HTML ed essi vengono visualizzati così come sono stati inseriti, l'utente potrebbe scrivere:

```
Che bel sito!<script>alert("Questo sito ti consiglia di visitare la pagina
                                        www.pincopalla.it")</script>
```

Con il risultato che a chiunque visiti la pagina dei commenti venga impartito un consiglio che apparirebbe come un consiglio del gestore del sito. Allo stesso modo è possibile inserire del codice JavaScript che comunica a siti esterni informazioni riservate sui visitatori o altre informazioni che potrebbero compromettere la sicurezza del sistema.

## 5.1.11 Come ci si difende?

Anche per il code injection è necessario prevedere un filtro; in questo caso non bisogna permettere che delle stringhe provenienti dal database appaiano come codice HTML oppure Javascript; l'alternativa è validare l'input iniziale non permettendo caratteri come ">" o "<" (validazione positiva) o filtri che li trasformino nelle corrispondenti codifiche standard. In questi casi i filtri a validazione negativa funzionano egregiamente quando il target è solo l'HTML, in quanto esistono apposite tabelle di transcodifica.

## 5.1.12 Impossibile prevedere l'uso dei dati!

Si potrebbe pensare di filtrare i caratteri che possono creare problemi di visualizzazione in pagine HTML ancora prima di inserirli nel database, usando un filtro "generale" che esegua l'escape sia dei caratteri dannosi per il database sia di quelli dannosi per la visualizzazione. Tale soluzione è da evitare: non si sa a priori (almeno in linea di principio) l'uso che si vuole fare di quanto viene inserito nel database. Potrebbe essere che anziché venir presentato sul Web esso

venga visualizzato in altro modo (console grafica di un'Applet, o altra tecnologia client, come Flash e plug-in specifici). Per questo motivo o si applicano validazioni positive in qualsiasi momento, purché antecedente al loro utilizzo, o si applicano i filtri a validazione negativa solo quando strettamente necessario e in maniera specifica rispetto a quanto si vuol realizzare.

## 5.2 AUTORIZZAZIONE E AUTENTICAZIONE

Nella quasi totalità delle applicazioni Web c'è almeno una parte ad accesso riservato o, per lo meno, esistono delle funzionalità di gestione la cui fruizione è riservata a degli utenti particolari (amministratori o, in genere, utenti con certi privilegi). In questi casi si parla di risorse la cui fruizione deve essere "autorizzata". Per autorizzare un utente lo si deve riconoscere, ovvero "autenticare". Vediamo quali problemi possono nascere e come realizzare delle soluzioni il più possibile riutilizzabili e che seguono gli standard J2EE.

### 5.2.1 Sicurezza: approccio dichiarativo o da programma

Esistono due modi sostanzialmente diversi di gestire la sicurezza: uno prevede di gestirlo a livello di container dell'applicazione (sia esso un servlet container come Tomcat o un application server come JBoss) e, in questo caso, si parla di approccio dichiarativo; il secondo modo consiste nel realizzare all'interno dell'applicazione uno o più moduli che si occupano della gestione delle problematiche di sicurezza. Cosa scegliere? Come sempre la risposta non è univoca ma ciascuna soluzione offre vantaggi e svantaggi. L'approccio dichiarativo permette di separare nettamente la gestione della sicurezza dal codice applicativo, permettendo di variare una parte senza intervenire sull'altra. Inoltre la stessa soluzione si può adottare per diverse applicazioni, senza necessità che queste siano realizzate in maniera tra loro simile. Purtroppo la modalità

di realizzazione è dipendente dal container (anche se ci sono tentativi di standardizzazione, alcuni dettagli possono sempre variare); inoltre per l'approccio dichiarativo si deve essere esperti amministratori dello strumento usato e non solo della scrittura di applicazioni Java. L'approccio programmatico permette, d'altro canto, di creare soluzioni ad hoc e tarate per le specifiche esigenze, ma a discapito dell'omogeneità di gestione tra applicazioni diverse. È anche vero che esistono modalità di realizzazione via codice che permettono un semplice riuso delle soluzioni e un significativo miglioramento della loro standardizzazione. Esiste infine una terza via che è a metà strada tra l'approccio dichiarativo e quello programmatico, che tenta di unire i vantaggi dei due mondi. Esso si basa su uno standard, chiamato JAAS (Java Authentication and Authorizazione Services) che permette di estendere l'infrastruttura di sicurezza messa a disposizione dal container.

## 5.2.2 Approccio dichiarativo: il caso di Tomcat

Tomcat si basa, anche per la gestione della sicurezza, sulla specifica delle Servlet; in tale specifica ogni utente può avere uno o più ruoli associati. Un ruolo è un'entità astratta a cui si può abilitare (o disabilitare) l'accesso a specifiche risorse. Ogni utente (o gruppo di utenti) può avere associato uno o più ruoli. Eventuali informazioni definite utilizzando l'approccio dichiarativo vanno inserite nel file che descrive l'applicazione, ovvero il file WEB-INF/web.xml. In tale file si dichiara quali ruoli hanno accesso a quali risorse. L'associazione tra ruoli e utenti è, invece, definita a livello di container. Esistono molti modi per definire l'associazione ruoli/utenti ma tutti sono definiti "realm". Tra essi è possibile farlo creando un file xml, usando un server LDAP o, ancora, una base dati relazionale e così via. In **Tabella 5.1** esempi di realm.
Si vedrà l'uso del real in memoria: si mostrerà com'è il file XML che accompagna la distribuzione standard di Tomcat che contiene utenti/ruoli e come realizzare applicazioni che ne facciano uso. È in-

| Tipo di Realm | Implementazione |
| --- | --- |
| Memory | Il documento conf/tomcat-users.xml contiene tutte le informazioni e queste sono lette allo start-up del container e mantenute in memoria |
| JDBC | Le informazioni sono memorizzate in una base dati relazionale acceduta usando un driver JDBC |
| DataSource | Anche in questo caso le informazioni sono su una base dati, ma sono accedute attraverso l'uso di un data source definito a livello JNDI |
| JAAS | Utilizza il framework Java Authentication & Authorization Service (JAAS) framework (secondo le specifiche J2EE) |
| JNDI | Le informazioni sono memorizzate su server LDAP, acceduto usando un provider JNDI |

**Tabella 5.1:** Tipi di realm riconosciuti da Tomcat

teressante notare che cambiando l'implementazione del real (intervenendo quindi a livello di container) non cambia alcunché a livello applicativo.

## 5.2.3 Dichiarare utenti e ruoli

Sotto la cartella /conf/ di installazione del Tomcat c'è il file tomcat-users.xml; aprirlo e inserire i seguenti tag XML:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
 <role rolename="tomcat"/>
 <role rolename="role1"/>
 <role rolename="manager"/>
 <role rolename="admin"/>
 <user username="tomcat"
  password="tomcat" roles="tomcat"/>
 <user username="role1"
```

```
  password="tomcat" roles="role1"/>
 <user username="both"
  password="tomcat" roles="tomcat,role1"/>
 <user username="admin"
  password="admin" roles="admin,manager"/>
</tomcat-users>
```

In pratica esso definisce questi realm: quattro ruoli (che sono tomcat, role1, manager ed admin) e quattro utenti, dove per ciascuno sono definiti username, password e ruoli associati (i ruoli sono inseriti come lista separata da virgole).
Nulla vieta di personalizzare questo file introducendo nuovi ruoli e utenti o modificando quelli esistenti. Si tenga presente che i realm hanno sempre valenza a livello di container (e quindi risultano definiti per ogni web application).
Per indicare a Tomcat che questo è il file da utilizzare nella definizione dei realm, è necessario verificare che nel file conf/server.xml ci sia il seguente tag nella sezione GlobalNamingResources (normalmente questa è la situazione di un'installazione standard, ma è sempre opportuno verificare che esista!):

```
<Resource name="UserDatabase" auth="Container"
 type="org.apache.catalina.UserDatabase"
 description="User database that can be updated and saved"
 factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
 pathname="conf/tomcat-users.xml" />
```

## 5.2.4 Associare ruoli a risorse per ogni webapp

A livello di web application si possono associare certi ruoli a risorse ben specifiche intervenendo sul file WEB-INF/web.xml dell'applicazione; ecco, per esempio, come creare due path applicativi "sicuri", il primo dei

quali accessibile dagli utenti con ruolo “tomcat” oppure “admin”, l’altro solo per utenti con ruolo “admin”:

```
<security-constraint>
 <web-resource-collection>
  <web-resource-name>Tomcat</web-resource-name>
  <url-pattern>/secure-tomcat/*</url-pattern>
 </web-resource-collection>
 <auth-constraint>
  <role-name>tomcat</role-name>
 </auth-constraint>
</security-constraint>

<security-constraint>
 <web-resource-collection>
  <web-resource-name>Tomcat</web-resource-name>
  <url-pattern>/secure-tomcat/*</url-pattern>
 </web-resource-collection>
 <web-resource-collection>
  <web-resource-name>Admin</web-resource-name>
  <url-pattern>/secure-admin/*</url-pattern>
 </web-resource-collection>

 <auth-constraint>
  <role-name>admin</role-name>
 </auth-constraint>
</security-constraint>
```

Si deve anche definire la modalità di riconoscimento dell’utente definito nei real; ecco, per esempio, come creare una http basic authentication:

```
 <login-config>
   <auth-method>BASIC</auth-method>
```

```
  <realm-name>Tomcat Manager Application</realm-name>
 </login-config>
```

Infine è necessario inserire la lista di tutti i ruoli precedentemente referenziati:

```
<security-role>
 <role-name>admin</role-name>
</security-role>
<security-role>
 <role-name>tomcat</role-name>
</security-role>
```

## 5.2.5 Single sign on

Nel caso di Tomcat di produzione in cui esistono molte (e diverse) applicazioni, è auspicabile che l'utente non debba autenticarsi per ognuna delle webapp. Esiste una funzionalità, chiamata "Single Sign On" per cui è possibile, sotto certe ipotesi, far sì che l'utente si autentichi una volta sola e riusi le sue credenziali per tutte le applicazioni a cui accede. Per informazioni su questo aspetto particolare è possibile far riferimento alla pagina http://localhost:8080/tomcat-docs/config/host.html#Single%20Sign%20On.

## 5.2.6 Accedere alle informazioni di autenticazione

Eventuali utenti (e ruoli) possono essere reperiti da codice Java usando le API messe a disposizione dalla classe HttpServletRequest (che rappresenta la richiesta attuale di una servlet, sia essa associata ad un metodo post che ad un metodo get); in particolare si possono utilizzare i metodi getRemoteUser per avere lo username utilizzato per l'autenticazione del client, getUserPrincipal che fornisce i ruoli associati (grazie ad

un oggetto della classe java.security.Principal) e, infine, il metodo isUserInRole che permette di verificare se l'utente appartiene ad un ruolo specifico. Negli esempi allegati al libro, le pagine index.jsp sotto le cartelle secure-tomcat/ e secure-admin/ non fanno altro che mostrare l'utente collegato:

```
<center>
Benvenuto <%= request.getRemoteUser() %>!
</center>
```

## 5.2.7 Approccio programmatico: filtri http

Quando si realizzano applicazioni complesse si rischia di realizzare troppe funzionalità all'interno di una o più classi, rendendo difficile sia la loro manutenzione (per aggiungere nuove funzionalità senza intaccare il corretto funzionamento di quelle esistenti) sia per la comprensione (soprattutto se si riprende lo sviluppo a distanza di molto tempo). Accanto alla possibilità di utilizzare pattern come l'MVC, visto in precedenza, l'ambiente Java ha dei componenti che implementano il pattern deco-

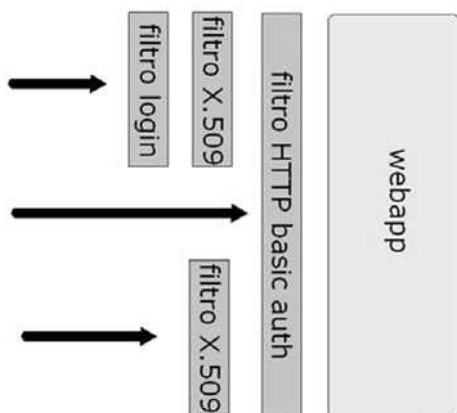



**Figura 5.2:** uso di filtri per l'autenticazione e autorizzazione degli utenti.

rator: i filtri (si riveda a questo proposito il Capitolo 1). Essi non sono altro che classi che implementano un'apposita interfaccia (Filter) e che si possono far rispondere "in catena" per specifiche url. In questo modo si possono realizzare filtri specifici che realizzano solo particolari funzionalità (un ottimo articolo di approfondimento sull'uso di Servlet e filtri è disponibile all'URL http://java.sun.com/developer/Books/javaserverpages/servlets_javaserver/servlets_javaserver09.pdf).
Un esempio potrebbe essere quello di realizzare dei filtri che si occupano dell'autenticazione e autorizzazione degli utenti, come mostrato in Figura 5.2.
Nella **Figura 5.2** si può notare che non è assolutamente vero che tutti i filtri entrano in gioco per tutte le richieste: infatti, attraverso il file WEB-INF/web.xml, si può specificare come ciascun filtro possa rispondere solo ad alcune URL (o a parte di esse). In questo modo ci sono un certo numero di filtri che autenticano l'utente (ciascun filtro acquisisce un solo tipo di credenziale) e un filtro si occupa dell'autorizzazione. Ecco, per esempio, un filtro che reperisce l'http basic authentication:



```
package it.ioprogrammo.filter;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class FiltroHttpBasicAuth implements Filter {
 public void init(FilterConfig filterC) throws ServletException {
 }
 public void doFilter(
   ServletRequest req,
   ServletResponse resp,
   FilterChain chain)
   throws IOException, ServletException {
   if (autorizzato(req))
     chain.doFilter(req, resp);
   else
```

```
    setBasicAuth((HttpServletResponse) resp);
  }
  public void destroy() {
  }
}
```

Ecco come si potrebbe implementare il metodo autorizzato:

```
  private boolean autorizzato(ServletRequest req) {
    String[] credenziali =
      getCredenziali((HttpServletRequest) req);
    return utenteValido(credenziali);
  }

  private boolean utenteValido(String[] utente) {
    return utente!=null &&
      utente.length==2 &&
      "test".equals(utente[0]) &&
      "test".equals(utente[1]);
  }
```

Ovviamente, in applicazioni reali, utenteValido andrà a reperire i dati su database, file system o altro. Ecco le classi che permettono il reperimento delle credenziali dalla request:

```
private static String[] getCredenziali(HttpServletRequest request) {
  String credenziali = null;
  try {
    sun.misc.BASE64Decoder decoder =
     new sun.misc.BASE64Decoder();
    credenziali =
      new String(
        decoder.decodeBuffer(
```

```
        request.getHeader("authorization"
          ).substring("Basic ".length())));
    int pos = credenziali.indexOf(":");
    if (pos < 0)
      return null;
    String[] s =
      new String[] {
        credenziali.substring(0, pos),
        credenziali.substring(pos + 1)};
    return s;
  } catch (Exception e) {
    return null;
  }
}
```

I seguenti metodi invece permettono di impostare un codice di errore di ritorno al client che gli indica la necessità di fornire credenziali di tipo Basic Authentication (quando il client riceve un errore 401 farà apparire all'utente una pop-up con richiesta di username/password):

```
private static void setBasicAuth(HttpServletResponse response)
  throws IOException {
  setStatus(response, 401, "WWW-Authenticate", "BASIC");
}

private static void setStatus(
  HttpServletResponse response,
  int cod,
  String key,
  String value)
  throws IOException {
  response.setStatus(cod);
  if (key != null && value != null)
```

```
    response.setHeader(key, value);
  else
    response.sendError(cod);
  response.flushBuffer();
}
```

## 5.2.7 JAAS e Tomcat

Tra i real di Tomcat c'è anche un real JAAS. JASS è un framework generale proposto dalla Sun per le problematiche di autenticazione e autorizzazione. In questo senso si può considerare JASSRealm una implementazione ibrida tra i metodi forniti da un specifico container e lo standard J2EE. Per i dettagli si può far riferimento alla documentazione di Tomcat e, più precisamente, alla pagina http://localhost:8080/tomcat-docs/realm-howto.html#JAASRealm (in realtà tale implementazione è un prototipo e, come tale, è soggetta a variazioni nelle future versioni).

## 5.2.8 Il Web e lo spam: CAPTCHA!

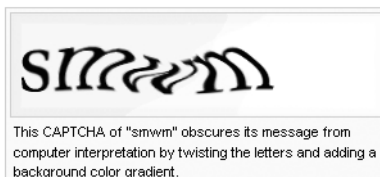
Le applicazioni Web non sempre hanno il vincolo che chi interagisce con



**Figura 5.3:** esempio di lettere "capibili" da una persona ma non da un programma.

esse debba essere un utente registrato. Si pensi ad un blog pubblico e non moderato: chiunque potrebbe inserire dei commenti ai post senza doversi registrare. Però questo tipo di applicazioni sono esposte ad un rischio: possono essere attaccate da programmi che inseriscono informazioni "spazzatura" (spam) in modo automatico. Un modo per evitarlo è quello di inserire delle informazioni che possono essere reperite (o dedotte) facilmente da un utente "reale", ma non da un programma. In questo momento esempi di questo tipo di "trappola" sono rappresentati da parole generate casualmente e le cui lettere appaiono distorte; una persona riesce comunque a distinguerle, mentre non riesce a farlo alcun programma (**Figura 5.3**). Per questo genere di "trappole" è stato coniato un acronimo: CAPTCHA, che sta per "Completely Automated Public Turing test to tell Computers And Humans Apart" (ovvero, con una traduzione un po' libera, "Test di turing completamente automatici per discriminare tra computer ed esseri umani"; il riferimento a Turing è dovuto a motivi storici: uno dei primi test per dire se un programma di Intelligenza Artificiale era sufficientemente "intelligente" era di far sedere una persona davanti ad un PC e farlo interagire, per mezzo di un'interfaccia, sia con un essere umano che con il programma: se la persona non era in grado di dire chi era il programma e chi l'essere umano, si poteva dire che il programma simulava il comportamento intelligente per cui era stato predisposto). Yahoo ha investito molte risorse per questo tipo di tecnologia. Si veda la pagina http://www.captcha.com per lo stato dell'arte della tecnologia. Chi volesse creare qualcosa di simile nelle proprie applicazioni può far uso del progetto Open Source chiamato JCaptcha e la cui home page è http://www.jcaptcha.net. L'utilizzo del tool è davvero molto semplice; per le prime informazioni essenziali si consiglia la lettura dell'articolo "5 minutes application integration tutorial" reperibile alla pagina http://forge.octo.com/jcaptcha/confluence/display/general/5+minutes+application+integration+tutorial. Un esempio d'uso di JCaptcha è stato realizzato sulla rivista IoProgrammo nel numero di luglio 2006 (le sue funzionalità venivano rese "pubbliche" realizzando un Web Service attraverso il framework Axis).

# ALTRI STRUMENTI E TECNOLOGIE

Accanto alle API standard di Java o a quelle che ne estendono le funzionalità di base, esistono numerosi tool e framework open source che affrontano e risolvono problemi specifici della programmazione di applicazioni Web. In questo capitolo verranno analizzate alcuni di essi, verrà evidenziato quando intervengono nel ciclo di vita del software e quali problematiche risolvono.

## 6.1 STRUMENTI PER IL TEST

Il test automatico delle applicazioni riveste particolare importanza per una serie di motivi: innanzi tutto avere una batteria di test permette di rieseguire rapidamente i test ad ogni modifica significativa, minimizzando il pericolo di aver introdotto nuovi errori (sia inserendo nuove funzionalità che modificando quelle esistenti; in quest'ultimo caso si parla di "regression test"); in secondo luogo fornisce al cliente un certo grado di sicurezza sul fatto che l'applicazione è stata verificata e ha modo di rieseguire i test codificati e di sincerarsi del loro successo. Ma cosa si intende per "test"? Esistono diversi aspetti; in particolare si identificano le seguenti problematiche:

**1)** test di modulo: ogni componente software (per Java esso può essere ogni singola classe) dovrebbe essere testato da solo per garantire che fa quello per cui è stato progettato. In una webapp secondo il modello MVC questa parte si focalizza sul test dei componenti del Model e, in minima parte, per la View;

**2)** test di integrazione: una volta che ogni componente è stato testato separatamente, è necessario assicurarsi che fra loro interagiscano in maniera corretta; quest'aspetto si chiama integrazione. In una webapp sviluppata con modello MVC si può considerare sia l'aspetto di integrazione fra diversi moduli del Model, sia l'interazione tra Model, View e Controller nel loro insieme;

**3)** test funzionale: questo test si concentra nell'interazione con un utente (reale o simulato) e verifica che sotto certi input l'applicazione fornisca output appropriati; in una webapp è la parte più complessa da rendere automatica;

**4)** test di stress (o test di carico): una webapp, in linea di principio, potrebbe essere acceduta contemporaneamente da un numero illimitato di utenti; al crescere del numero di utenti si ha un maggior consumo di risorse e un inevitabile rallentamento dell'intera applicazione. Questo tipo di test consiste nel verificare il comportamento simulando delle situazioni "limite" (ovvero considerando un numero massimo di accessi contemporanei e verificando che comunque l'applicazione risponda correttamente ed in tempi ragionevoli).

Per ciascuna di queste problematiche esistono strumenti differenti. Ecco, in sintesi, quelli a disposizione in ambiente Java. Resta inteso che molto spesso i framework presentati sono così ricchi di funzionalità che possono essere utilizzati per diversi tipi di test; essi sono presentati unicamente in una categoria solo perché essa è quella per cui sono stati pensati o per cui forniscono strumenti di gestione più appropriati. Si rimanda alla documentazione di ciascuno strumento per approfondirne l'uso.

## 6.1.1 Test di modulo: JUnit

JUnit è considerato il tool di riferimento per i test di modulo (sito web *http://www.junit.org*). L'ultima release (la 4.0) richiede Java versione 5; infatti, a differenza delle altre versioni, fa uso delle annotazioni (in particolare di @Test) e non più dell'ereditarietà (insieme a convenzioni sui nomi e alla reflection) comportando un vero e proprio cambio di direzione rispetto alle versioni precedenti. Eseguire il download del file compresso dal sito e met-

tere nel classpath il file junit-4.0.jar ivi contenuto. Ora è possibile far uso dello strumento e, in particolare, creare una classe che esegue i test  (per eseguire dei test su una classe, è necessario creare una ulteriore classe che invochi i metodi della classe originaria e che verifichi che il risultato si quello desiderato). I metodi che eseguono il test saranno contrassegnati dalla annotazione @Test; per esempio ecco la classe che esegue il test di modulo della classe it.ioprogrammo.jspAdvanced.jmx.Sessioni:

```
package it.ioprogrammo.jspAdvanced.junit;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import it.ioprogrammo.jspAdvanced.jmx.Sessioni;
public class SessioniTest {
 @Test public void testSessioniAddDel(){
  Sessioni s = new Sessioni();
  long num1 = s.getActiveSessions();
  s.addSession();
  s.delSession();
  long num2 = s.getActiveSessions();
  assertEquals(
   "Il numero di sessioni non è corretto", num1, num2);
 }
}
```

In pratica è stato inserito un metodo che crea una nuova istanza della classe Sessioni, prende il numero di sessioni attive, ne crea una nuova e ne elimina una e, infine, verifica che il numero di sessioni attive sia rimasto invariato. Come mandare in esecuzione il test? Non avendo il classico metodo main che rende la classe invocabile, si deve far uso di un “runner” apposito. Al momento di scrivere, JUnit 4.0 dispone unicamente di un program-

ma testuale per l'esecuzione dei test (chiamato TestRunner). È probabile che al momento di leggere queste righe sia stato fornito anche un qualche runner grafico (infatti quasi tutti gli strumenti di sviluppo hanno uno strumento grafico per l'esecuzione dei test). Se così non fosse, come non lo è al momento di scrivere il libro, si può usare un adattatore creato appositamente per far uso degli strumenti grafici esistenti; per usarlo basta inserire il seguente metodo:

```
public static junit.framework.Test suite() {
  return new
   junit.framework.JUnit4TestAdapter(
     SessioniTest.class);
}
```

In Figura 6.1 l'esempio della sua esecuzione utilizzando Eclipse 3.1. Si può notare che il test è andato a buon fine.

## 6.1.1 Predisporre i test

Esistono anche altri tipi di annotazioni che servono per preparare l'ambiente prima di eseguire i test e dopo (per liberare eventuali risorse o ripristinare la situazione di partenza). Infatti i metodi contrassegnati con @Before sono eseguiti prima di eseguire uno dei test (quindi se ci sono due test, essi vengono eseguiti due volte: una prima volta prima del primo test, poi prima di eseguire il secondo test); quelli contrassegnati con @After vengono eseguito dopo l'esecuzione di ciascun test (nell'esempio di due test, anch'essi vengono eseguiti due volte); se si vuole che alcuni metodi vengano eseguiti una sola volta prima di iniziare il test bisogna contrassegnarli con @BeforeClass (tali metodi vengono eseguiti una ed una sola volta e poi iniziano i test). Viceversa se si vuole che un metodo venga eseguito una sola volta alla fine di tutti i test esso va annotato con @After-

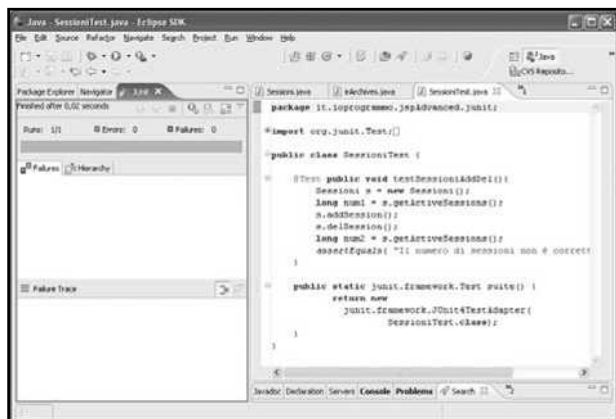



**Figura 6.1:** Esempio di esecuzione di un test con JUnit.

Class. Nel caso si usino le notazioni @BeforeClass e @AfterClass i metodi devono essere dichiarati static.
Ecco una possibile modifica all'esempio precedente, in cui l'istanza di Sessioni viene creata all'inizio di ogni test (e messa a null alla fine) ma fuori dal metodo stesso:

```
public class SessioniTest {
 private Sessioni s;
 @Before public void inizializza(){
  s = new Sessioni();
 }
 @After public void finalizza(){
  s = null;
 }
 @Test public void testSessioniAddDel(){
  long num1 = s.getActiveSessions();
  s.addSession();
  s.delSession();
  long num2 = s.getActiveSessions();
```

```
  assertEquals(
   "Il numero di sessioni non è corretto", num1, num2);
 }

 public static junit.framework.Test suite() {
  return new
    junit.framework.JUnit4TestAdapter(
      SessioniTest.class);
 }
}
```

Ma come (e quando) si scrivono i test di modulo? Il consiglio che ritengo più appropriato è quello dato dalle metodologie agili (vedi *http://agilealliance.org/*): scrivere un numero significativo di test ancor prima di realizzare il modulo (in Java un modulo lo si può identificare in una classe) realizzando solo i metodi con un corpo vuoto (pertanto la classe iniziale non fa nulla se non definire i metodi che espone). È ovvio che finché la classe non viene completata i test non avranno successo. Però, via via, sempre più test avranno successo e, completata la classe, tutti dovrebbero andare a buon fine. Questo non significa (purtroppo) che la classe sarà priva di errori. Ogni volta che si trova un nuovo errore procedere come segue: tentare di creare un nuovo test che lo identifichi (il test, pertanto, dovrà fallire). Una volta corretto l'errore verificare la soluzione rieseguendo la batteria di test; quando tutto è sistemato ci sarà un nuovo test. In questo modo i test crescono man mano viene eseguito il debug. Attenzione alla falsa sicurezza data dalla scrittura di test non appropriati; a questo proposito si veda l'articolo "JUnit antipatterns - How to resolve", in cui di dice che la parte difficile non è imparare ad usare JUnit, ma è creare "buoni" test di modulo; l'articolo è consultabile alla pagina *http://www-128.ibm.com/developerworks/opensource/library/os-junit/?ca=dgr-lnxw07JU-*

*nite*. Sintetizzando le idee dell'autore ecco a cosa bisogna prestare attenzione:

**1)** non creare mai meno di due test diversi per la verifica di correttezza di un metodo;
**2)** usare anche test dove l'input non fa parte del range ammesso;
**3)** creare anche test con valori "al limite", ovvero vicini al limite inferiore e superiore del range di validità;
**4)** non scrivere mai test troppo complessi (magari più complessi del codice che testano!); in questi casi non si sa, in presenza di errori, se l'errore è imputabile al codice da testare o al test stesso.

Accanto a JUnit esistono altre proposte; alcune sono nate proprio per superare alcuni dei (presunti o reali) limiti di JUnit. Per esempio TestNG ("Test Next Generation", http://testng.org) e JTiger (http://jtiger.org/). Molte idee presentate da tali tool sono state recepite proprio dalla versione 4.0 di JUnit, mostrando come la ricchezza delle proposte porti ad un miglioramento dei tool esistenti. Esistono poi tool che estendono JUnit per compiti specifici. Un esempio è DBUnit che permette di testare quelle parti di programma che interagiscono con il database. Il progetto, la cui pagina principale è http://www.dbunit.org/, permette di settare lo stato del db prima e dopo i test ma è sempre consigliato avere a disposizione più basi di dati (una creata appositamente per i test, magari una per ogni sviluppatore che esegue il test, e che sia sempre sincronizzata con la base dati di produzione).

## 6.1.2 Test di integrazione: Cactus

Cactus (*http://jakarta.apache.org/cactus/*) è un framework che permette il test di applicazioni all'interno di un container (per esem-

pio all'interno di Tomcat). Eseguito il download del framework (va bene la forma binaria, se non si vuole compilare i sorgenti), è necessario predisporre un insieme di test (chiamati TestCase) che sono molto simili ai TestCase per JUnit ma, in questo caso, sono specifici per gli oggetti da testare (è possibile scrivere test per servlet, taglib, JSP ma anche filtri): questo permette, nel caso di applicazioni MVC, di testare separatamente le diverse componenti e, in linea di principio, consente lo sviluppo separato di parti di logica (i bean del Modeler) da quelle di presentazione (JSP della parte View). La versione di Cactus analizzata è la 1.7.2: essa fa ancora uso di JUnit 3.8.1; pertanto i test sono scritti come classi che estendono delle classi di base, le quali sono specifiche per gli oggetti da testare; ecco i casi per, rispettivamente, servlet, tag JSP e filtri:

```
public class EsempioTestServlet extends ServletTestCase {
 // implementazione del test
}

public class EsempioTestTag extends JspTestCase {
 // implementazione del test
}

public class EsempioTestFiltro extends FilterTestCase {

 // implementazione del test
}
```



JUnit stesso è usato come client per l'esecuzione dei test. Cactus mantiene dei metodi per l'inizializzazione dell'ambiente prima di eseguire i test e metodi per pulirlo al termine come fa JUnit, ma offre anche dei metodi per inizializzare/ripulire l'ambiente del client che esegue i test.

## 6.1.3 Test funzionale: HttpUnit, Watij e WebTest

HttpUnit (sito *http://httpunit.sourceforge.net/*) permette il test di siti Web emulando l'accesso alle pagine Web senza usare un browser. L'emulazione consente di esaminare le pagine Web sia come testo che come XML (DOM), sia usando strutture che contengono eventuali form, tabelle e link. È possibile anche usare JavaScript contenuto nelle pagine, la basic http authentication, i cookies e simulare la navigazione (anche con frame). Permette di astrarre dal tipo di campo di inserimento della form (può essere sia un radio button, una combo box che un campo di inserimento), offrendo un minimo di flessibilità per la scrittura dei test nei confronti dell'implementazione specifica dell'oggetto da testare. Tra le altre cose il software permette il test delle servlet al di fuori di un servlet container. L'oggetto principale è WebConversation che incapsula le funzionalità (simulate) di un browser; ecco, per esempio, come accedere al sito Web dell'autore del libro e stampare il contenuto della home sullo standard output:

```
WebConversation webc = new WebConversation();
WebResponse webr =
 webc.getResponse( "http://ivenuti.altervista.org" );
System.out.println( webr.getText() );
```

Dal WebResponse è anche possibile reperire informazioni specifiche (applet, DOM, HTML ma anche insiemi specifici come links, frames, tabelle e così via) memorizzate in oggetti appositi che, a loro volta, forniscono numerose primitive di accesso alle informazioni contenute.
Una delle caratteristiche di HttpUnit è quella di integrarsi con il framework Cactus, visto precedentemente, per realizzare test sia di integrazione che funzionali.
Un altro tool per i test funzionali è Watij (http://watij.xwiki.com/):

esso permette test funzionali utilizzando un browser vero e proprio e interagendo con esso (al momento è supportato unicamente IE su piattaforme Windows). In pratica il browser viene modellato da un oggetto e su di esso è possibile caricare pagine, invocare metodi che simulino l'interazione con la pagina stessa (submit su form, inserimento di testo, invocazione di metodi JavaScript e così via) e verificare come vengono generate le pagine successive. Lo svantaggio più evidente nell'utilizzare questo tipo di tool è che i test sono quasi sempre molto sensibili alle variazioni, anche minime, di dettagli non sempre visibili. Pertanto test di questo tipo sono, di solito, difficili da mantenere quando il sistema evolve e vanno riscritti o personalizzati per le nuove modifiche. Un po' più vicino al modo di "vedere" dell'utente è il tool WebTest della Canoo (sito Web http://webtest.canoo.com/); in esso si descrivono dei casi d'uso (use cases ) con una sintassi XML; ecco, per esempio, come accedere ad una pagina *http://127.0.0.1:8080/JspAspettiAvanzat i/index.htm* e verificare che il titolo corretto:

```
<project name="TestAppLibro" basedir="." default="main">
 <property name="webtest.home"
  location="/home/ioprogrammo/webtest" />
 <import file="${webtest.home}/lib/taskdef.xml"/>
 <target name="main">
  <webtest name="TestAccesso">
   <config
     host="127.0.0.1"
     port="8080"
     protocol="http"
     basepath="JspAspettiAvanzati" />
   <steps>
     <invoke
       description="Accedi alla prima pagina"
```

```
      url="index.jsp" />
    <verifyTitle
     description="Titolo corretto"
     text="Esempi del libro 'JSP - Aspetti Avanzati' -
                                              Autore: Ivan Venuti"
    />
  </steps>
 </webtest>
</target>
</project>
```

### 6.1.4 Test di carico: JMeter

JMeter (*http://jakarta.apache.org/jmeter/*) può essere utilizzato sia per test funzionali ma anche per test di carico. In particolare può connettersi sia a pagine Web che ad altre risorse (basi di dati, server FTP, JNDI, SOAP, file e così via). Può simulare un numero qualunque di richieste (sia contemporanee che differite) e mostrare il comportamento della risorsa sotto test con grafici e report. Può gestire cookie, diversi tipi di autenticazione, far uso di proxy server e molto altro. Dispone di una console grafica che permette sia di impostare i test che di visualizzare i risultati in forma di grafici o report (si veda la Figura 6.2).
Essendo un client vero e proprio non si integra con le applicazioni di cui esegue il test, ma si limita a eseguire l'accesso ad esse. Questo permette di eseguire test di carico per tutti i siti Web, indipendentemente dal fatto che siano sviluppati con Java o con altre tecnologie.

## 6.2 OLTRE LE JSTL: LIBRERIE DI TERZE PARTI

Sul sito di Jakarta ci sono, tra le altre cose, anche delle librerie

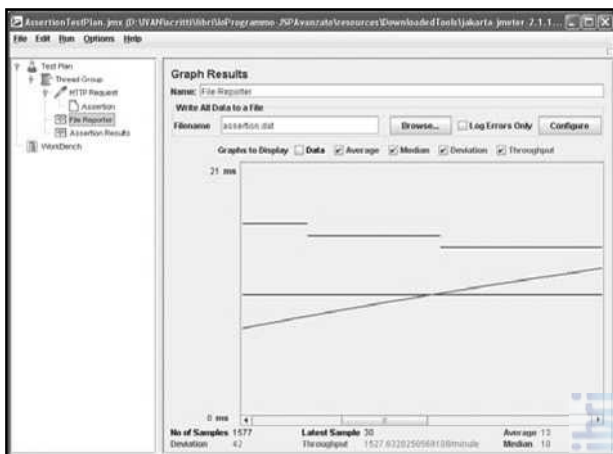



**Figura 6.2:** Esempio di grafico per un test di carico eseguito con JMeter.

di tag aggiuntive (il nome del progetto è "Taglibs"). Infatti, prima di divenire librerie standard, le JSTL erano state realizzate nel progetto Jakarta all'interno del sito come librerie di utilità generale. Alcune sono divenute parte della distribuzione originale, altre sono rimaste librerie di utilità fornite unicamente da Jakarta. Pertanto vale la pena di fare una visita al sito per verificare se alcune di esse possono risultare utili per i propri programmi. La pagina di riferimento è *http://jakarta.apache.org/taglibs/*. (Figura 6.3).
Esistono poi delle librerie "specializzate", che risolvono compiti specifici.

## 6.2.1 Usare un gestore di log

I log sono particolari file per la memorizzazione delle informazioni necessarie sia per il monitoraggio del funzionamento di un'applicazione, sia per risalire ad eventuali problemi di funzio-

namento o anomalie che si sono presentate (per esempio richieste che non si possono soddisfare, tentativi di login non autorizzati e così via).

**Figura 6.3:** Taglibs: le librerie di tag del progetto Jakarta.

Anziché utilizzare lo standard output (dove andrebbero a finire tutte le informazioni di tutte le applicazioni, creando una notevole confusione) è possibile ricorrere ad opportuni pacchetti specializzati per la gestione dei log.
Log4J è, con tutta probabilità, lo strumento più utilizzato per questo scopo. Alla pagina *http://logging.apache.org/log4j/docs/download.html* si può accedere al download della libreria e ad ulteriori estensioni, una delle quali è proprio una tag library.

## 6.2.2 Layout a griglia e ordinamento

Quando si usano grandi quantità di dati, è comune presentarli su griglie (stile foglio di calcolo). Una funzionalità utile consiste nella possibilità di ordinare i dati per colonne, esportarli in altri

formati e così via. Un progetto interessante che permette tutto questo è la tag library <display:*>, scaricabile dal sito http://displaytag.sourceforge.net/.

## 6.2.3 Gestione dei grafici

Prima o poi un'applicazione dovrà mostrare dati in forma di grafici (a torta, a barre, e così via) sia per mostrare statistiche che per sintetizzare informazioni numeriche. Una libreria di tag che ne semplifica la creazione è CeWolf, il cui sito di riferimento è http://cewolf.sourceforge.net/. In **Figura 6.4** alcuni tipi di grafici supportati.

## 6.2.4 Siti e risorse

In Tabella 6.1 un elenco di siti utili per trovare tag libraries adatte ai propri scopi.

# 6.3 TAPESTRY

Tapestry (*http://jakarta.apache.org/tapestry/*) è un framework piuttosto particolare. Giunto alla release 4.0 esso è a pieno di-

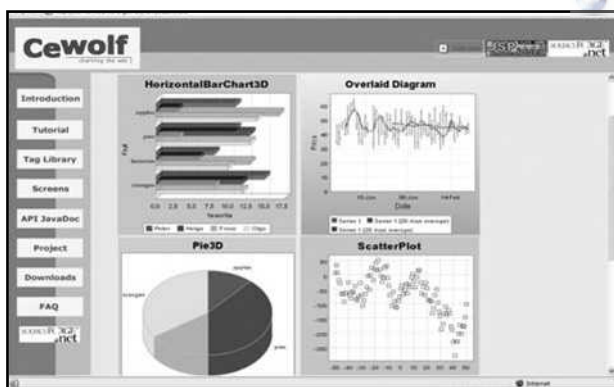



**Figura 6.4:** Alcuni tipi di grafici supportati da CeWolf.

ritto tra i progetti più diffusi per la costruzione di applicazioni Web in Java (non a caso fa parte del progetto Jakarta di Apache). Esso tenta di mascherare le usuali problematiche di costruzione

| Sito Web | Descrizione |
|---|---|
| http://jakarta.apache.org/taglibs/ | Le già citate Taglibs del progetto Jakarta |
| http://java-source.net/open-source/jsp-tag-libraries | Collezione di librerie Open Source |
| http://www.jspin.com/home/tags | Librerie di tag categorizzate per utilizzo e ambito d'applicazione |
| http://jsptags.com/tags/index.jsp | Riferimenti e link a numerose librerie di tag |

**Tabella 6.1:** Portali per tag libraries.

di webapplications nascondendo i dettagli legati alla tecnologia di accesso (http, con le relative richieste, risposte, url di accesso...) e proponendo un'architettura dove esistono solo componenti (oggetti Java); tali componenti possiedono dei metodi ascoltatori (listener methods) che si occupano di interagire con i componenti software (come EJB o JavaBean) e sono responsabili di scegliere le pagine risultato (pagine che si basano su dei template HTML e che contengono al loro interno una composizione di componenti, attraverso un approccio dichiarativo). In pratica il focus è sui componenti (component-centric) non sulle operazioni (operation-centric). Accanto ai numerosi componenti standard esistono componenti forniti dalla comunità di sviluppatori (Tapestry Contrib Library). Tra le novità della release 4.0 c'è il supporto nativo alle specifiche Portlet.
A mio avviso lo svantaggio più evidente di questo framework è di una sintassi completamente proprietaria per la creazione delle pagine template; infatti, per scelta, Tapestry non fa uso di tag JSP (lo standard nell'architettura J2EE).

## 6.4 QUANDO NON BASTA TOMCAT

In molti contesti di produzione, Tomcat può essere affiancato da altri strumenti per sopperire ad alcune sue mancanze o per migliorarne le prestazioni. Vediamo alcuni casi concreti.

### 6.4.1 Integrazione con Apache

Apache è, ad oggi, il maggior Web Server utilizzato, grazie alle sue ottime caratteristiche di robustezza, velocità e configurabilità. Tomcat ha un supporto limitato alle operazioni eseguite da programmi cgi-bin e non è la scelta migliore per servire grosse quantità di contenuti statici. In questi casi si può installare un Apache e demandare a Tomcat solo le richieste relative a Web Application scritte in Java.
La configurazione dei due strumenti in modo tale che siano messi in comunicazione non è particolarmente complessa. I dettagli per la realizzazione di quanto detto sono reperibili sull'ottimo documento pubblicato alla pagina *http://www.greenfieldresearch.ca/technical/jk2_config.html*.

### 6.4.2 Cenni sugli Application Server e JBoss

JBoss (*http://www.jboss.com*) rappresenta lo stato dell'arte degli application server Open Source. Gli Application server possiedono caratteristiche evolute per realizzare applicazioni di classe enterprise. Tra le caratteristiche di maggior interesse c'è senz'altro la possibilità di fornire supporto per gli EJB (Enterprise JavaBean). Pertanto un Application Server non è indispensabile per utilizzare una Web Application che faccia uso solo di Jsp, Servlet e JavaBean. Anzi: i più comuni Application Server (JBoss in primis) utilizzano al loro interno un Tomcat come Web Container. Il resto dell'infrastruttura dell'Application Server offre caratteristiche aggiuntive e avanzate
Pertanto il consiglio è quello di avvicinarsi a JBoss o a qualsia-

si altro Application Server unicamente per utilizzare (o scoprire) le feature avanzate in caso di effettiva necessità.

## 6.4.3 JBoss: un'architettura a strati

L'architettura di JBoss è divisa in strati, ciascuno dei quali gestisce funzionalità ben specifiche e ha responsabilità chiare e ben definite (ulteriori dettagli sono presenti alla pagina *http://www.jboss.com/products/ jbossas/architecture*, Figura 6.5):

**Microkernel Layer**: è il "cuore" del server e utilizza la tecnologia JMX per fornire funzionalità di deploy delle applicazioni e caratteristiche per la gestione del ciclo di vita di una funzione (comprese funzionalità avanzate per il class-loading)

**Services Layer**: appena sopra lo strato precedente, si situano i diversi servizi base di JBoss AS (quali gestione delle transazioni, servizi e-mail, gestione della sicurezza, pooling di oggetti e così via). Le tante configurazioni disponibili, utilizzate in fase di

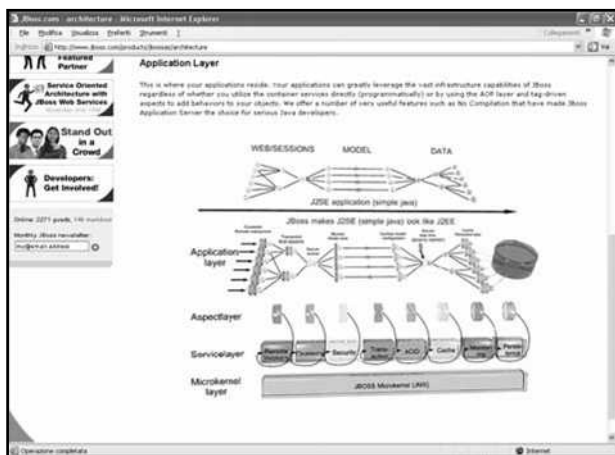



**Figura 6.5:** L'architettura a strati di JBoss.

avvio di JBoss, si riferiscono a diversi tipi di servizi abilitati. Tali configurazioni agiscono proprio su questo livello: esse permettono di calibrare opportunamente solo i servizi necessari. È anche possibile realizzare servizi aggiuntivi, che si collocano in questo strato; essi vanno resi disponibili come pacchetti di tipo SAR.

**Nota**: SAR: Service ARchive; sono dei formati proprietari di JBoss per il deploy di nuovi servizi. Essi possono essere installati anche durante il funzionamento di JBoss, rendendo l'ambiente particolarmente flessibile anche in ambienti di produzione, dov'è sconsigliato fermare un qualsivoglia server anche solo per aggiornarne l'architettura e le funzionalità.

**Aspect Layer**: si basa su un nuovo paradigma di programmazione, l' Aspect-Oriented Programming (AOP). Questo strato è accessibile anche da quello successivo (applicativo) permettendo l'uso dei suoi costrutti e della programmazione "tag-driven" anche nelle applicazioni utente;

**Application Layer**: l'ultimo strato è quello applicativo ovvero dove risiedono le applicazioni sviluppate. Tali applicazioni si possono basare su funzionalità esposte dagli strati inferiori e si possono usare per realizzarne di nuove.

## 6.4.4 Services Layer: cosa c'è "dentro"

Il service layer gestisce, in maniera modulare, le seguenti funzionalità:

- Pieno supporto allo standard J2EE/EJB e implementazione degli standard ivi definiti;
- EJB 3.0: piena implementazione della nuova specifica per gli Enterprise JavaBean
- Meccanismo di cache per il miglioramento delle performance delle applicazioni Enterprise;

- Meccanismo di logging (utilizzando log4j);
- Sicurezza (compatibile con la specifica JAAS, ma anche moduli personalizzati);
- JNDI server;
- Servizi RMI/IIOP (invocazione remota via RMI);
- CORBA/IIOP (invocazione di servizi remoti con CORBA);
- Servizi per il clustering (con meccanismi di fail-over, load-balancing e di deploy distribuito); fornisce supporto anche per la gestione di repliche e di fail-over per le sessioni http;
- Un database embedded (hypersonic);
- Un modulo per realizzare Web Services (basato su Axis);
- JMS 1.1 (Java Messaging Service) per lo scambio di messaggi (sincroni/asincroni)
- JTS (Java Transaction API) per tutti quei contesti applicativi dov'è necessario l'uso di transazioni;
- JCA (supporto di J2EE Connector API).

Inoltre viene utilizzato un Web Container per gestire le applicazioni Web scritte in Java; in particolare, JBoss 4 utilizza Tomcat 5 in forma "embedded"; questo si traduce in un sottoinsieme di funzionalità (e possibilità di configurazione) rispetto alla versione stand-alone.
L'editor: JBoss Eclipse IDE integra perfettamente Eclipse e JBoss AS, fornendo anche un supporto per la generazione e la scrittura di XDoclet nonché Wizard per lo sviluppo di applicazioni J2EE, compresi EJB e WebServices, ma anche editor per pagine JSP e documenti XML.

## 6.5 SERVIZI DI HOSTING GRATUITI PER SERVLET E JSP

Esistono numerosi servizi di hosting commerciali. Invece ce ne sono pochissimi gratuiti. Infatti, a differenza di altre tecnologie

come il PHP o il Perl, le JSP sono poco supportate da chi offre spazio web e risorse in maniera gratuita. Per esempio esiste *http://www.myjavaserver.com* (Figura 6.6) che offre supporto alla tecnologia J2EE e 5 Mb di spazio su disco, nonché numerosi servizi aggiuntivi (database, server SMTP, librerie di terze parti e così via) ad oltre 40000 sviluppatori.
Quando ci si registra con un certo nome utente, si ha a disposizione una pagina che inizia con *http://www.myjavaserver.com/~nomeUtente/* (il carattere "strano" è una tilde: se non compare sulla tastiera la si può ottenere tenendo premuto il tasto [Alt] e digitando, in sequenza, i tasti [1], [2], [6]).
C'è una convenzione sui nomi dei package per garantire la si-

**Figura 6.6:** hosting gratuito per Servlet e JSP su http://www.myjavaserver.com

curezza tra utenti diversi: la prima parte del nome del package deve essere uguale a quella del nome utente. Sembra una cosa di poco conto ma, purtroppo, questo si applica anche a classi fornite da terze parti! Questa è, con tutta probabilità, l'unica caratteristica negativa del servizio.

## 6.6 DIREZIONI PER IMPARARE ALTRE TECNOLOGIE O STANDARD

Java è un mondo sconfinato, soprattutto per quanto riguarda le applicazioni server di classe enterprise ma anche quelle che sono considerate le best-practice nello sviluppo delle applicazioni Web. Un solo libro non potrebbe descrivere adeguatamente tutte le tecnologie; in questo caso si ritiene utile almeno inquadrarle nel loro giusto contesto e a dare delle indicazioni per approfondirle in caso di necessità.

### AJAX

AJAX non è propriamente né una tecnologia né un framework: infatti propone un'architettura dove l'elaborazione da parte del client si concretizza in chiamate asincrone fatte via Javascript e XML. L'idea, estremamente interessante, applica il pattern MVC anche lato client. Infatti permette l'uso di codice JavaScript che interagisce direttamente con il server, senza basare l'interazione sulle usuali operazioni di post e get del protocollo http. Per approfondirne l'uso di può partire dall'articolo "Ajax: A New Approach to Web Applications" di Jesse J. Garrett, presente alla pagina *http://www.adaptivepath.com/publications/essays/archives/000385.php*; infatti è in quest'articolo che viene evidenziata e descritta l'idea base.
Interessanti risorse ed esempi si trovano anche alle pagine *http://java.sun.com/developer/technicalArticles/J2EE/AJAX/* ("Asynchronous JavaScript Technology and XML (AJAX) With Java 2 Platform, Enterprise Edition" di Greg Murray), *http://developer.mozilla.org/en/docs/AJAX* (sono presenti sia una introduzione che una selezione di articoli su Ajax), *http://www.ibm.com/developerworks/library/j-ajax1* ("Ajax for Java developers: Build dynamic Java applications"), *http://www.cgisecurity.com/ajax/* (risorse e link a tutorial sulla tecnologia e problemi di sicurezza), *http://www.ajaxmatters.com* (risorse e ar-

ticoli). Chi invece fosse interessato a tool di sviluppo può visitare la pagina *http://www.backbase.com* (Backbase AJAX tools). AjaxTags, una libreria di tag per utilizzare Ajax nelle proprie pagine JSP, è disponibile per il download alla pagina *http://ajaxtags.sourceforge.net/*.

**EJB**

Gli Enterprise JavaBean sono i componenti server-side di riferimento delle applicazioni Enterprise in ambiente J2EE. Per un loro studio non si può non partire dalla pagina ufficiale della Sun (http://java.sun.com/products/ejb/) dove, oltre a trovare articoli e risorse, si può venire aggiornati sull'evoluzione della specifica. Ottima anche la documentazione messa a disposizione da JBoss (si veda la pagina http://www.jboss.com/products/ejb3); inoltre è reperibile, alla pagina http://www.theserverside.com/books/wiley/masteringEJB/, la terza edizione del libro "Mastering Enterprise Java Beans".

**JMS**

Tra le varie architetture middle-tier, una ha dimostrato di essere particolarmente robusta e versatile: quella dei messaggi (sincroni e asincroni). Java implementa una struttura di messaggistica attraverso delle API di alto livello, chiamate Java Messaging Services. Per un'introduzione alla tecnologia si può scaricare "Java Message Service Tutorial" della Sun alla pagina *http://java.sun.com/products/jms/tutorial/*.

**Portlet**

Quando si realizzano portali Web si hanno delle necessità peculiari per la gestione delle informazioni e l'organizzazione dei contenuti. Java ha introdotto una specifica, le Portlet per l'appunto, per realizzare componenti web utilizzabili all'interno di appositi container come se fossero dei plug-in. In questo modo

è possibile assemblare le pagine utilizzando le portlet come componenti base. Questo permette sia la modularità dei componenti sia il loro riuso. La specifica è scaricabile dalla pagina *http://www.jcp.org/en/jsr/detail?id=168*.
Per degli articoli introduttivi alla tecnologia si vedano i link *http://www.mokabyte.it/2003/06/jportlet-1.htm*, *http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-portlet.html*.

**Web Services**

Sempre più, nelle applicazioni Enterprise, ci si trova nella necessità di utilizzare dati e servizi provenienti da applicazioni remote. Quando tali applicazioni remote sono scritte nella stessa tecnologia, si può utilizzare un protocollo di comunicazione specifico per tale tecnologia. È il caso di DCOM e di DCOM+ per la piattaforma Microsoft o di RMI per le applicazioni Java. Esistono poi degli standard, come CORBA, in linea di principio generali e non specifici (CORBA è utilizzabile da un qualsiasi linguaggio orientato agli oggetti). Ultimamente stanno avendo sempre più successo i Web Services, o servizi Web. Tale successo nasce dall'uso di tecnologie standard e già ampiamente adottate (quali XML per la rappresentazione dei dati, XML Schema per la loro validazione, tcp/ip per l'infrastruttura di trasporto) e dalla capacità di essere una tecnologia facilmente implementabile con strumenti e toolkit disponibili nei più diffusi linguaggi ed essere davvero svincolata dai linguaggi in uso.
In ambiente Java esistono numerosi toolkit e implementazioni ma quella che merita senz'altro un'attenzione particolare è Axis (sito di riferimento per la versione 1 è *http://ws.apache.org/axis*, la versione successiva è reperibile alla pagina *http://ws.apache.org/axis2* ) un progetto completamente Open Source dalle buone caratteristiche architturali e facile da usare. Chi volesse approfondire l'argomento può consultare il testo "Web Services – Guida pratica" ([Venuti, 2005]).

# 6.7 VALIDARE LE PAGINE HTML (W3C)

In ambito internazionale da molti anni la problematica dell'accessibilità dei siti Web è un argomento di discussione e di approfondimento da parte degli addetti ai lavori. Per accessibilità di un sito Web si intende la caratteristica di un sito web di essere realizzato in aderenza agli standard e di non presentare particolarità che ne impedirebbero la fruizione (per esempio il sito web non deve richiedere l'uso di una versione aggiornata del browser, non deve richiedere la presenza di colori, ma deve rendere accessibili i contenuti anche senza immagini e/o colori e così via).

In Italia è stata approvata una legge (la cosiddetta "legge Stanca") che impone (quindi obbliga!) a tutti i nuovi siti della pubblica amministrazione di essere accessibili secondo linee guida indicate dal Ministero. Tale obbligo non vale per i cittadini e le aziende private; per tali soggetti la legge si limita a consigliare di adottare soluzioni accessibili.

In ogni modo si può intuire come quest'aspetto possa rivestire una primaria importanza per siti web che vogliano presentarsi come professionali e in linea con tali suggerimenti, magari con tanto di "certificazione".

La certificazione rispetto alla legge Stanca è possibile ma piuttosto costosa.

Viceversa è possibile ottenere un riconoscimento ufficiale (attraverso l'uso di un'icona standard) per quelle pagine che passano la validazione del W3C (organismo internazionale che definisce gli standard per il Web).

Infatti, accedendo alla pagina http://validator.w3.org/ è possibile sottoporre le pagine alla validazione del codice HML (secondo i diversi standard) e alla pagina http://jigsaw.w3.org/css-validator/ si possono validare le pagine per quanto concerne i CSS.

In **Figura 6.7** il sito dell'autore contenente i pulsanti che atte-

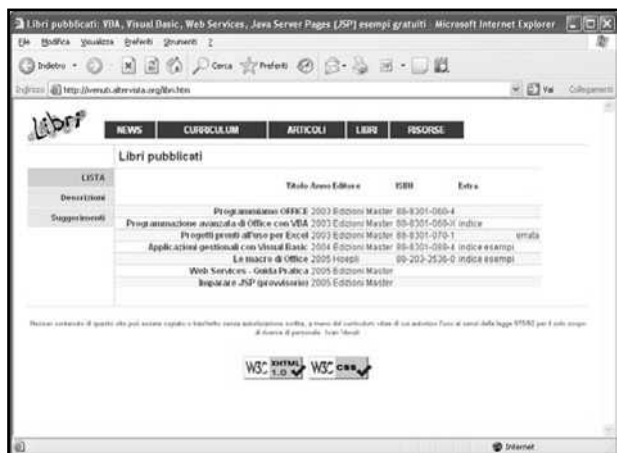



**Figura 6.7:** Il sito dell'autore aderente agli standard XHTML e CSS.

stano che le pagine del sito sono valide secondo gli standard XHTML e CSS.
Esistono anche altri validatori; in **Tabella 6.2** alcuni utilizzabili direttamente on-line.

| Sito Web | Descrizione |
|---|---|
| http://validator.w3.org/ | "W3C Markup Validation Service" |
| http://jigsaw.w3.org/css-validator/ | Servizio, sempre del W3C, per la validazione dei fogli di stile a cascata (CSS) |
| http://www.htmlhelp.com/tools /validator/ | WDG (Web Design Group) HTML Validator |
| http://onlinewebcheck.com/ | Validatore gratuito che si basa su un prodotto commerciale (CSE HTML Validator Lite) |
| http://www.doctor-html.com/RxHTML/cgi-bin/single.cgi | Doctor HTML |

**Tabella 6.2:** Validatori di codice html.

# SERVLET 2.5 E JSP 2.1

Attualmente la specifica di riferimento per ambienti di produzione è la 2.4 per le servlet e la 2.0 per le JSP. La prossima release è la 2.5 per le servlet (JSR 154, *http://jcp.org/en/jsr/detail?id=154*) e la 2.1 per le JSP. Tomcat 6 sarà la versione del Servlet Container di riferimento per le nuove specifiche delle Servlet/JSPche, al momento di andare in stampa, è ancora in beta. Come si vedrà le nuove versioni delle specifiche introducono interessanti semplificazioni per la scrittura di componenti lato server e una reale integrazione tra le tecnologie JSF e JSP. Per poter provare le nuove specifiche è necessario installare un server che le implementi. Tomcat 6 sarà la versione del Servlet Container di riferimento per le nuove specifiche versione che, al momento di andare in stampa, è ancora in beta.

## 7.1 QUALE TOMCAT?

Spesso si ha il dubbio: quale Tomcat installare? In Tabella 7.1 le majour release e una loro spiegazione (per l'ultima versione stabile di ciascuna release si faccia riferimento al sito Web ufficiale di Tomcat: *http://tomcat.apache.org/*). In tabella sono elencate solo le versioni successive alla 3. Infatti sono le uniche versioni rese pubbliche. Le altre erano utilizzate internamente da Sun prima che il progetto fosse reso Open Source e donato alla Apache Software Foundation.

**Attenzione**
Utilizzare Tomcat 6 è utile solo se si ha il desiderio di verificare le nuove caratteristiche delle specifiche oppure per contribuire al suo sviluppo. Infatti tale servlet container non è stabile e non possiede, al momento, un'adeguata documentazione che ne descrivono le nuove potenzialità.

| Specifiche | Tomcat | Descrizione |
|---|---|---|
| Servlet 2.5, JSP 2.1, JSF 1.2 | 6.0 | Ultima release ma in versione beta. Sconsigliato l'uso in ambienti di produzione. Vengono introdotte le annotazioni. Per questo motivo per usare questa versione si deve usare Java 5 (o successivi). |
| Servlet 2.4, JSP 2.0 | 5.5 | Ultima release stabile e attualmente quella di cui è consigliato il download per nuove installazioni. |
| Servlet 2.4, JSP 2.0 | 5.0 | Pur implementando le stesse specifiche della 5.5 ha, rispetto ad essa, non possiede molte delle migliorie architetturali introdotte in essa; per questo è utile passare al più presto alla versione 5.5. Per questa versione è presente un XML Schema per la validazione del file web.xml |
| Servlet 2.3, JSP 1.2 | 4.1 | Benché si ritrovi ancora presso molti siti in produzione, è auspicabile passare al più presto alla versione 5.5. è in questa versione della specifica che sono stati introdotti i filtri. |
| Servlet 2.2, JSP 1.1 | 3.3 | È stata la prima versione di Tomcat. Release oramai considerata obsoleta. La specifica prevedeva, per la prima volta, la possibilità di deploy delle webapp in formato war. |



## 7.2 LE NUOVE SPECIFICHE

Ma, in concreto, quali sono le nuove caratteristiche delle nuove specifiche? Eccone alcune:

- Un generale riallineamento delle tecnologie JSP/JSF (in precedenza presentavano alcuni problemi dovuti ad una non ottimale integrazione);
- Supporto per le annotazioni (questo è coerente con tutte le evoluzioni della piattaforma Java, si pensi per esempio alle nuove specifiche EJB 3);
- Unified EL: l'expression language ha una nuova sintassi che ne permette l'uso in maniera sintatticamente coerente sia nelle JSP che nelle JSF;
- Modalità che semplificano la scrittura dei contenuti del file web.xml;

- trimDirectiveWhitespaces: nuova direttiva per le JSP che minimizza gli spazi bianchi generati;
- impossibilità per una web application di ridefinire le librerie necessarie al container (per esempio non è più possibile usare una versione antecedente né di JSP né di JSP!).

## 7.2.1 Unified EL

EL è stato introdotto nelle specifiche JSP 2.0 per essere usato nelle JSTL. Subito dopo è stato adottato anche dalle JSF ma, in tale contesto, era necessario estenderlo per supportare altre caratteristiche quali la valutazione differita (per poter usare il risultato della valutazione nei componenti a cui si voleva assegnare dei valori è necessario "differire" la valutazione dell'espressione ad una delle fasi del ciclo di vita di una JSF che non coincide con quelli delle JSP), uso di espressioni sia per leggere che per settare valori (quindi usare la stessa espressione sintattica per indicare, in casi diversi, una lettura o un assegnamento). Queste nuove caratteristiche hanno portato ad ovvi conflitti nelle pagine che fanno uso sia di JSP che JSF. L'unica possibilità è stata quella di creare un nuovo EL, integrato e, pertanto, unificato. Ecco la nascita di "Unified EL"; linguaggio che è stato introdotto come standard a partire da JSP 2.1. Ma quali sono le caratteristiche di Unified EL? Per espressioni come

```
<fmt:formatNumber value="${nomeBean.Proprieta}"/>
```

sono sempre valutate immediatamente e possono essere usate solo per leggere valori (non per settarli).
Queste espressioni sono le usuali espressioni EL usate anche in JSP 2.0. Invece se si considera:

```
<h:inputText id="nomeProprieta" value="#{nomeBean.nomeProprieta}" />
```

C'è un diverso comportamento in base al fatto che il tag sia presente in una pagina su cui avviene una nuova richiesta oppure nel caso in cui si provenga dalla stessa pagina in seguito ad un submit della form. Nel primo caso l'espressione viene valutata da JSF nella fase "render response". Nel secondo caso il valore sulla request viene letto, validato e assegnato alla omonima proprità del bean "nomeBean". In maniera analoga è possibile inserire dei metodi che verranno valutati in un momento differito, come può essere il caso di un validatore su una proprietà:

```
<prefix:inputText id="nomeProprieta"
 value="#{nomeBean. nomeProprieta}"
 validator="#{nomeBean.validateNomeProprieta}"/>
```

Ma che accade se le pagine esistenti contengono tag o valori che possiedono i caratteri "#{" che, ovviamente, nella versione precedente non erano riservati? La prima soluzione è fare l'escape del primo carattere usando un carattere backslash: "\#{". Altrimenti è possibile ricorrere ad un nuovo attributo dell'elemento page:

```
<%@page ... deferredSyntaxAllowedAsLiteral="true" %>
```

## 7.2.2 Un risolutore... personalizzato e problemi di migrazione

Con Unified EL è possibile usare un meccanismo che indica quando e come risolvere le espressioni differite usando un'architettura a plug-in. In pratica è possibile usare un risolutore standard o crearne uno personalizzato.

Per maggiori informazioni su come crearne uno ex novo si

può far riferimento alla pagina
*http://today.java.net/pub/a/today/2006/03/07/unified-jsp-jsf-expression-language.html*.
Chi è abituato ad usare delle proprie librerie di tag potrebbe desiderare di usare il nuovo Unified EL all'interno dei propri tag. La migrazione non è particolarmente complessa; chi è interessato può far riferimento al tutorial della Sun accessibile alla pagina
*http://java.sun.com/products/jsp/reference/techart/unifiedEL.html*.

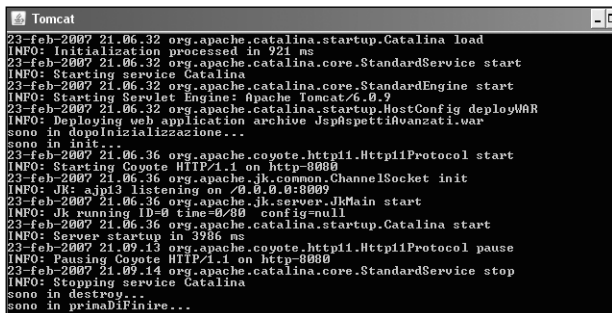
## 7.2.3 Nuove annotazioni

Alcune nuove annotazioni permettono di definire a livello di sorgente Java quello che prima doveva essere specificato nel file web.xml; altre annotazioni semplificano le operazioni da fare. Per esempio è possibile riferirsi ad un DataSource definito a livello di container usando la nuova annotazione @Resource:

```
@Resource javax.sql.DataSource qualeDb;
// ...
Connection con = qualeDb.getConnection();
// ...
```

In maniera analoga è possibile riferirsi ad oggetti EJB del container usando l'annotazione @EJB (nelle specifica 3.0 esistono anche altre annotazioni: @PersistenceContext, @PersistenceContexts, @PersistenceUnit, @PersistenceUnits). Altre annotazioni riguardano il ciclo di vita degli oggetti e, in particolare, sono utili per le servlet: @PostConstruct permette di "etichettare" un metodo affinché venga eseguto subito dopo il costruttore dell'oggetto (per una servlet è il metodo init()? Non proprio! Si vedrà fra un attimo cosa accade… ); in maniera si-

mile @PreDestroy permette di indicare che un metodo deve essere invocato quando l'oggetto sta per essere distrutto. È importante che questi metodi non restituiscano dati (void) e che non sollevino eccezioni. Ecco un esempio di una servlet che fa uso di queste ultime due annotazioni:

```
package it.ioprogrammo.jspAdvanced.Servlet25;

import javax.annotation.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class NewFeatures extends HttpServlet implements Servlet{

 public void init(ServletConfig config) throws ServletException {
  System.out.println("sono in init...");
 }

 @PostConstruct
 public void dopoInizializzazione(){
  System.out.println("sono in dopoInizializzazione...");
 }

 public void destroy() {
  System.out.println("sono in destroy...");
 }

 @PreDestroy
 public void primaDiFinire(){
  System.out.println("sono in primaDiFinire...");
 }

}
```

Provando ad eseguire lo startup e successivo shutdown del server ci si accorge che l'ordine di invocazione è dopoInizializzazione, init per lo startup, mentre destroy, primaDiFinire per lo shudown (**Figura 7.1**).

```
Tomcat
23-feb-2007 21.06.32 org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 921 ms
23-feb-2007 21.06.32 org.apache.catalina.core.StandardService start
INFO: Starting service Catalina
23-feb-2007 21.06.32 org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.9
23-feb-2007 21.06.32 org.apache.catalina.startup.HostConfig deployWAR
INFO: Deploying web application archive JspAspettiAvanzati.war
sono in dopoInizializzazione...
sono in init...
23-feb-2007 21.06.36 org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on http-8080
23-feb-2007 21.06.36 org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
23-feb-2007 21.06.36 org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/80  config=null
23-feb-2007 21.06.36 org.apache.catalina.startup.Catalina start
INFO: Server startup in 3986 ms
23-feb-2007 21.09.13 org.apache.coyote.http11.Http11Protocol pause
INFO: Pausing Coyote HTTP/1.1 on http-8080
23-feb-2007 21.09.14 org.apache.catalina.core.StandardService stop
INFO: Stopping service Catalina
sono in destroy...
sono in primaDiFinire...
```

**Figura 7.1:** L'ordine delle stampe aiuta a comprendere l'ordine di invocazione dei vari metodi.

Esistono poi ulteriori annotazioni. Particolarmente interessante è l'uso dell'annotazione @DeclareRoles: permette di definire dei ruoli, usati nella webapp, per la sicurezza in maniera analoga ai tag <security-role> presenti nel file web.xml.

## 7.2.4 Novità nella scrittura del file web.xml

Sono state appena introdotte le annotazioni; esse sono un comodo modo per semplificare la scrittura di applicazioni. Purtroppo questo ha un impatto negativo sui tempi di inizializzazione delle applicazioni. Per questo c'è modo di specificare nel web.xml che la webapp che descrive non ne fa uso grazie all'attributo metadata-complete:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                 http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   version="2.5"
   metadata-complete="true">
 <!-- altro -->
</web-app>
```

Il significato di impostare a True l'attributo metadata-complete è che questo descrittore (ed eventuali altri descrittori associati) sono completi, ovvero non hanno bisogno di esaminare le annotazioni per recuperare ulteriori informazioni di deploy. Nel caso non sia specificato o sia specificato un valore False, il tool deve esaminare le classi che compongono l'applicazione per ricercare eventuali annotazioni. Questo significa rallentare sulla fase di inizializzazione di ogni applicazione. Per questo è importante specificare questo attributo se si vuole mantenere una programmazione "tradizionale" che non faccia uso delle nuove annotazioni.

**Attenzione**
Nel caso venga specificata come versione della webapp la 2.4, il valore di default dell'attributo metadata-complete è True. Questo è coerente con il fatto che in tale specifica non era contemplato l'uso delle annotazioni.

Nel caso si voglia associare un filtro a tutte le servlet (attenzione: non a tutte le url, ma alle servlet definite!) è possibile usare il carattere "*":

```
<filter-mapping>
  <filter-name>QualeFiltro</filter-name>
```

```
  <servlet-name>*</servlet-name>
</filter-mapping>
```

Queste e altre caratteristiche possono essere analizzate nel dettaglio reperendo l'XML Schema associato alla nuova specifica; esso è reperibile all'indirizzo *http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd*.

## 7.3 TOMCAT 6

Tomcat 6 avrà, oltre al supporto delle nuove specifiche, una serie di interessanti miglioramenti sia in termini di funzionalità che di performance. In particolare sarà possibile utilizzare Web Services per l'amministrazione del container (compresi servizi per il deploy). Indagando sul file RELEASE-NOTES che accompagna l'ultima release (per chi scrive la 6.0.9) si legge che sono include le seguenti librerie:

- annotations-api.jar: è il package che permette l'uso delle annotazioni;
- catalina.jar: l'implementazione di Tomcat Catalina;
- catalina-ant.jar: i task Ant per Tomcat Catalina;
- catalina-ha.jar: High availability package;
- catalina-tribes.jar: Apache Tribes, un modulo per la Group communication;
- commons-logging-api.jar: funzionalità di log (Commons Logging API 1.0.x);
- el-api.jar: implementazione di Unified EL, usando le API versione 2.1;
- jasper.jar: Jasper 2 Compiler and Runtime per la compilazione ed esecuzione delle webapp;
- jasper-el.jar: implementazione di Jasper 2 EL;
- jasper-jdt.jar (Eclipse JDT 3.2 Java compiler): permette di

usare Tomcat con il solo JRE (e non con un JDK completo!);
- jsp-api.jar (JSP 2.1 API): implementazione delle nuove specifiche per le JSP;
- servlet-api.jar (Servlet 2.5 API): implementazione nuove specifiche Servlet;
- tomcat-coyote.jar: connettori (per la comunicazione con altri server) e altre classi di utilità;
- tomcat-dbcp.jar: connection pool che si basa sul progetto Commons DBCP.

Come si può notare, tra le nuove caratteristiche, c'è l'uso di un nuovo modulo per il clustering, denominato Apache Tribes (http://tomcat.apache.org/tomcat-6.0-doc/tribes/introduction.html. Apache Tribes, che è un messaging framework, può essere usato anche come framework standalone nelle proprie applicazioni, permettendo la creazione di ambienti Java distribuiti.



**Attenzione**
Tomcat resta il servlet container di riferimento per le specifiche Servlet/JSP. Ma se si desidera un container che implementi tutte le tecnologie J2EE si deve far riferimento ad un altro progetto Open Source: GlassFish. La pagina di riferimento del progetto è http://glassfish.dev.java.net. Altro progetto estremamente interessante è Apache Geronimo (http://geronimo.apache.org/), prossimo alla versione 1.2, con supporto delle specifiche Servlet 2.5/JSP 2.1, Annotations 1.0, JAF 1.1, JavaMail 1.4 e numerose altre tecnologie (tra le quali JPA, JTA, JMS 1.1, JCA 1.5, JACC 1.1).

## 7.3.1 Moduli opzionali

Per i deploy esiste un nuovo modulo: il Tomcat Client Deployer

(TCD). Esso non è distribuito con la versione core di tomcat ma è presente come modulo separato (apache-tomcat-6.0.x-deployer). Esso si basa su script Ant per il deploy "a caldo" di nuove webapp (comprese la validazione e il corretto packaging in un file war).
È prevista anche l'implementazione delle specifiche SJR 109 per i Web Services. Però, al momento di scrivere, è disponibile come patch separata e non integrata nella distribuzione standard (*http://fabien.carrion.free.fr/tomcat/tomcat-6.0-patch-webservices.diff.bz2*); per i dettagli dell'installazione della patch si veda http://fabien.carrion.free.fr/Tomcat.html (sullo stesso sito è disponibile un'ulteriore patch per realizzare Single-SignOn sul nuovo Tomcat).

## 7.4 CONCLUSIONI

È probabile che non tutti gli sviluppatori sentano l'esigenza di passare alle nuove specifiche Servlet e JSP. Il fatto che Tomcat 6 non abbia raggiunto ancora una maturità tale da poter essere usato in ambienti di produzione ne ritarderà l'adozione. In ogni modo anche le nuove caratteristiche sono indispensabili solo se si vogliono adottare sia le JSF che le JSP. Infatti solo con le nuove specifiche sono stati risolti numerose e subdole incompatibilità e problematiche dovute ad una loro mancata integrazione. Se, invece, le JSF rappresentano una tecnologia imprescindibile per la propria attività, è bene prestarvi fin d'ora attenzione e valutare l'immediata adozione dei nuovi standard.

# APPENDICE MATERIALE SUL WEB

Di seguito vengono indicate utili risorse per apprendere (o approfondire) le basi di Java e della programmazione con le JSP, basi considerate acquisite all'interno del libro. Gran parte degli argomenti sono propedeutici ai diversi aspetti affrontati nel libro. Coloro che hanno delle lacune su questi aspetti sono invitati a leggere le risorse gratuite accessibili in rete per apprendere queste basi (uno sguardo a tali risorse è auspicabile anche chi, pur conoscendone le basi, vorrebbe approfondire uno o più di questi aspetti).

## JAVA

Prima di affrontare lo studio di applicazioni Web è necessario avere delle buone conoscenze sul linguaggio Java. In **Tabella A.1** alcune risorse per conoscere o approfondire tale linguaggio.

| Sito Web | Descrizione |
|---|---|
| http://java.sun.com | Il sito di riferimento per Java |
| http://www.java-net.it/ | Sito del libro "Java mattone dopo mattone", da cui è possibile scaricare un draft del libro stesso |
| http://www.mindview.net/Books/TIJ/ | Ottimo libro su Java di Eckel: "Thinking in Java"; dalle basi del linguaggio agli aspetti avanzati |
| http://www.techbooksforfree.com/java.shtml | Raccolta di libri gratuiti disponibili in rete |

**Tabella A.1:** Siti Web per conoscere il linguaggio Java

## JSP & SERVLET

Ovviamente non poteva mancare una lista di siti Web per approfondire lo studio della tecnologia JSP e delle Servlet. In **Ta-**

**bella A.2** sono presentate alcune tra le migliori risorse disponibili.

| Sito Web | Descrizione |
|---|---|
| http://java.sun.com/products/jsp/ | Pagina di riferimento per le JSP |
| http://java.sun.com/products/servlet/ | Pagina di riferimento per le Servlet |
| http://pdf.coreservlets.com/ | Download della prima edizione del libro "Core Servlets and Javaserver Pages" |
| http://www.jspinsider.com/ | Risorse per JSP e Servlet |
| http://www.javaworld.com/ | Tantissimi articoli e tutorial dedicati al mondo Java |
| http://www.latoserver.it/java/ | Un portale (in italiano) dedicato alle tecnologia server-side |
| http://www.html.it/jsp/ | Lezioni e approfondimenti su JSP e Servlet |
| http://www.roseindia.net/jsp /jsp.htm | Tutorial e documentazione su JSP e JSTL. |
| http://tomcat.apache.org/ | Dalla home page di Tomcat, accesso ad una nutrita serie di documenti utili per chi sviluppa Servlet e JSP |

**Tabella A.2:** Siti Web per JSP e Servlet.



# HTML E TECNOLOGIE CONNESSE

Sviluppare pagine Web presuppone che si abbia un'ottima conoscenza sia dell'HTML che di altre tecnologie, quali JavaScript, CSS e problemi legati all'accessibilità delle applicazioni. In **Tabella A.3** alcune risorse per ciascuna di queste tecnologie.

| Sito Web | Descrizione |
|---|---|
| http://www.html.it | Numerosi tutorial su tutto quanto concerne lo sviluppo di applicazioni Web, sia per la parte client (html, css, Javascript) che server (java, jsp) |

| | |
|---|---|
| http://www.w3schools.com/html /default.asp | HTML Tutorial dalla W3Schools (sul sito molti altri tutorial su numerose tecnologie) |
| http://werbach.com/barebones/ | Bare Bones Guide to HTML |
| http://zircon.mcli.dist.maricopa.edu/writinghtml_it/ | Writing HTML; traduzione in italiano |
| http://www.htmlgoodies.com/ | Tutorial su html e tecnologie correlate |
| http://www.htmlprimer.com/ | Tutto su html, css, xml |

**Tabella A.3:** Risorse per conoscere HTML e altre tecnologie lato client.

# BIBLIOGRAFIA

[Bayern, 2002] "*JSTL in Action*", , S. Bayern, Manning Publications, 2002

[Flanagan, 2005] ""*Java in a Nutshell*", 5a Edizione, D. Flanagan, O'Reilly, 2005

[Gamma et Al., 1994] "*Design Patterns: Elements of Reusable Object-Oriented Software*", E. Gamma et Al. Addison-Wesley, 1994

[Hall, Web] "*Core Servlets and JavaServer Pages*", 1a Edizione, M. Hall, disponibile per il download alla pagina http://pdf.coreservlets.com/

[Hall&Brown, 2003] "*Core Servlets and JavaServer Pages, Vol. 1: Core Technologies*", M. Hall, L. Brown, 2a Edizione, Prentice Hall PTR, 2003

[Hall, 2001] "*More Servlets & JavaServer Pages*", M. Hall, Pearson Education, 2001

[Mann, 2005] "*JavaServer Faces in Action*", K. D. Mann, Manning Publications Co., 2005

[Tarquini & Ligi, 2002] "*Java Mattone dopo Mattone*", Tarquini M., Ligi A, 2003 (draft del libro scaricabile dal sito http://www.java-net.it/)

[Tichy, 1998], "*A catalogue of General-Purpose Software Design Patterns. University of Karlsruhe*", W.F. Tichy, Karlsruhe, 1998,

http://wwwipd.ira.uka.de/~tichy/publications/Catalogue.doc

[Venuti, 2005] "*Web Services – Guida Pratica*", I. Venuti, Edizioni Master, 2005

[Venuti, 2006] "*Imparare JSP*", I. Venuti, Edizioni Master, 2006

[Zimmer, 1995], "*Relationships between Design Patterns - Pattern Languages of Program Design*", W. Zimmer, 1995 Addison-Wesley

**i libri di**

# LAVORARE CON JSP

**Autore:** Ivan Venuti

**EDITORE**
Edizioni Master S.p.A.
Sede di Milano: Via Ariberto, 24 - 20123 Milano
Sede di Rende: C.da Lecco, zona ind. - 87036 Rende (CS)

**Realizzazione grafica:**
Cromatika Srl
C.da Lecco, zona ind. - 87036 Rende (CS)

**Art Director:** Paolo Cristiano
**Responsabile grafico di progetto:** Salvatore Vuono
**Coordinatore tecnico:** Giancarlo Sicilia
**Illustrazioni:** Tonino Intieri
**Impaginazione elettronica:** Francesco Cospite

**Servizio Clienti**
☎ **Tel. 02 831212 - Fax 02 83121206**
@ **e-mail: customercare@edmaster.it**

**Stampa:** Grafica Editoriale Printing - Bologna

Finito di stampare nel mese di Marzo 2007